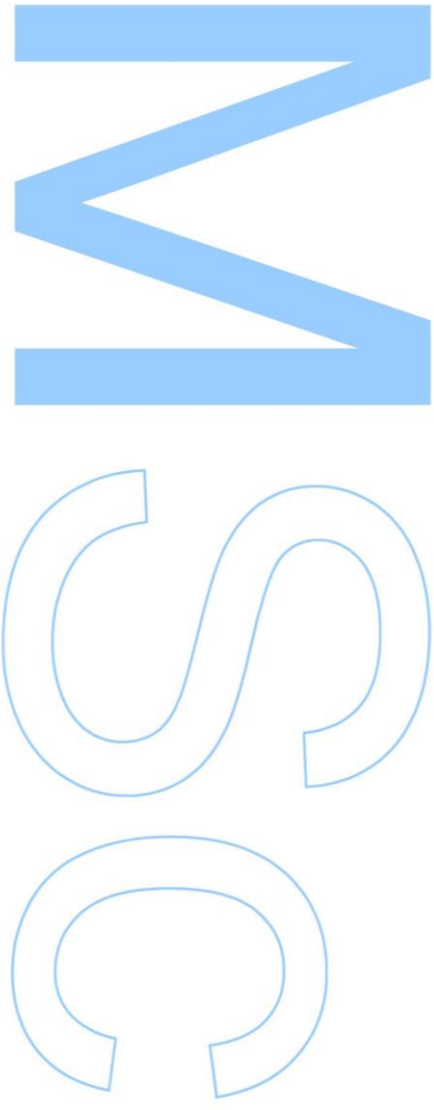
USB connection vulnerabilities on Android smartphones



André Fernando Lopes Pereira
Mestrado integrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2014

Orientador
Prof. Doutor Pedro Brandão, DCC-FCUP

Co-orientador
Prof. Doutor Manuel Correia, DCC-FCUP

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____

Agradecimentos

Em primeiro lugar, quero agradecer a ambos orientadores, Professor Pedro Brandão e Professor Manuel Correia. Desde que tomaram a decisão de me aceitarem para tese, passando por todos os processos que foram necessários para a conclusão da mesma. Sempre se mostraram disponíveis para atender as dificuldades encontradas e também para as ultrapassar.

Quero agradecer aos meus pais, pois sempre acreditaram na minha educação, não há maneira de expressar a gratidão que sinto por eles.

Por fim, mas sem menos importância, quero agradecer aos meus amigos, pois também foram um elemento muito importante antes, durante e depois da tese. Para eles um obrigado por me aturarem.

Resumo

Neste trabalho enumerámos uma série de vulnerabilidades relacionadas com ligação USB do Android, em especial vulnerabilidades relacionadas com a personalização feita pelos fabricantes de dispositivos Android.

Uma das principais vulnerabilidades que descobrimos resulta da personalização feita ao Android por alguns fabricantes, em que torna possível enviar comandos de serie AT proprietários por USB. Estes comandos AT proprietários que estendem as funcionalidades tradicionais dos comandos AT. Com isto é possível injetar código no telemóvel, que altera a partição boot. O que permite concluir vários objetivos, um deles obter acesso root.

Estas vulnerabilidades encontradas permitem-nos desenvolver vários ataques, que depois são ordenamos segundo a sua perigosidade e cujo objetivo é construir uma hierarquia de ataques de modo a corresponder o ataque mais eficaz que funcione com o dispositivo conectado

Elaboramos depois uma prova de conceito conjuntamente com um cenário de ataque concentrados na USB do Android. De seguida com alguns anti-virus Android, para verificar se de alguma forma estes detetam ou previnem o ataque.

Por fim desenvolvemos uma aplicação capaz de ajudar a mitigar estes ataques. A aplicação necessita permissão root para ser ativada. Caso não tenha seja permitido acesso root a aplicação apenas notificará o utilizador dos perigos inerentes a ligações USB.

Abstract

In this work, we enumerate a series of vulnerabilities in the USB connection of Android, specially vulnerabilities related with the customization done by the Android manufacturers.

The crux of the discovered vulnerabilities is a consequence of the vendor customization of Android, where the serial AT commands processed by the cellular modem are extended to allow other functionalities. With this, we are able to flash a boot partition on the smartphone and obtain several objectives, including root access.

These found vulnerabilities allow us to develop several attack vectors, which are ordered according with its efficacy, the objective in building an hierarchy that is able to produce the maximum efficacy attack according to the device.

We develop a proof of concept and an attack scenario specially designed for USB attacks on Android. We made some tests with popular anti-virus for Android, to examine if they can prevent or detect the attacks.

Finally, we develop an application capable of mitigating the discovered vulnerabilities, unfortunately the application needs root access to be activated. In case there is no root access, we prompt a warning to the user, notifying the dangers he may be in.

Contents

List of Figures

Chapter 1

Introduction

The extended functionalities available in smartphones are crucial to explaining its success, which lead to a fast transition over traditional phones. Functionalities such as phone banking, Global Positioning System (GPS), e-mail, together with the old functionalities like phone calling and Short Message Service (SMS) make the smartphone essential to our daily lives and ease our existence.

One major characteristic that leads to the proliferation of smartphones is its portability. Such characteristic together with the functionalities mentioned, inevitably lead to an increase of privacy invasion. Especially attacks that exploit the physical attack surface on the smartphone.

Attacks such as exploiting the Near Field Communication (NFC) [8, 9] or the Universal Serial Bus (USB) connection are uncommon attacks, therefore these types of attacks to a user are very threatening. A continuous research of the practical threat that these attacks pose should be placed, which is the principal aim employed in this dissertation.

1.1 Problem description

1.1.1 Android OS

The Android platform initially developed by Android inc. and bought by Google in 2005 [10], is an open-source operating system based on Linux kernel for mobile phones. The first Android phone launched was in 2008 and we have now seen nineteen versions

launched.

Android comprises 80% of the worldwide market share [11], making it the biggest player on the smartphone business. Vendor customization is one of the advantages of the Android ecosystem, allowing vendors to customize the operating system. Vendors add another layer of software over the standard release of the operating system. This is a double-edged sword, since it could introduce serious security breaches in the vendor modification of the operating system.

Independently of the Android version, vendors' customization could lead one product to be vulnerable and another not. Attackers could exploit different kind of attacks, over the many different products that run Android. According to a recent study, vendor customization accounts for 60% [12] of the vulnerabilities found in the Android ecosystem. Due to this, the vendor customization layer, is the primary target for the research of vulnerabilities in this dissertation.

1.1.2 Physical attacks threat

Security experts often overlook the security of USB attacks. Indeed, we have seen an increasing trend in the usage of physical attacks such as USB. The recent Stuxnet [13] attack is a good example. It was of most notorious successfully operated attack that exploited several vulnerabilities in the Windows operating system. Most notably however was how it spread, from system to system via USB pen drives. The worm efficacy of Stuxnet is notorious. First, it would spread over USB, infecting the host computer. Then it would spread over that computer's network, infecting the USB drives that are connected. With this strategy, Stuxnet was able to infect over 100 000 computers worldwide [13], however 99,9% of those infected computers where not the target. The target of the attack was PLC (Programmable Logic Controllers), which regulated the centrifuges separating nuclear material in Iranian nuclear power plants. Not just any PLC, the target was precisely a PLC at a specific nuclear facility in Iran.

Attacks carried out through USB, provide a very effective way of infecting large organizations. Such organizations usually tend to look more to the defense against attacks on their servers and the entire infrastructure accessible online. Forgetting that employees have access inside the infrastructure and can very well compromise it by bringing an

already infected USB pen drive.

Employees may intentionally conduct fraudulent activities as detailed in [14] and directly steal important information from the company. Employees can use the USB charging cable connected to a corporate computer, appearing to be charging the smartphone. In reality, they are using it to transfer data to the smartphone. This way of stealing important information from the company is very distinct, because charging the smartphone is a very common activity and no one would notice it.

Another possibility is that employees are completely unaware of the attack. They carry the USB pen drive from home to the company or any other intermediate location, they could be targeted in one of those locations and then later carry malicious code directly to the company, like in the case of Stuxnet. Employees' homes PC and firewalls are much less secure and the attackers know how to take advantage of these circumstances [15].

1.2 Motivation

During the master degree, computer security was always an interest of mine, leading me to appreciate the complex systems necessary to maintain our privacy in the digital world. It also made me realize how very easily just a little flaw, a bug, a badly implemented security principal, can break these systems.

Proximity attacks are a good example of this. USB connection's security, because often overlooked, contain smaller flaws, which have huge impact on system security.

This led me to pursuit the topic of this dissertation and investigate what kind of flaws might be explored in the USB connection of Android.

1.3 Proposal

In this dissertation, we explore vulnerabilities in the Android operating system that are associated with its USB connection. We investigate known and unknown vulnerabilities, both in the general release of Android but especially in the vendor customizations. On those we were able to inject malicious code, by flashing a compromised boot partition.

First we build a list of attacks and with that list, we order the attacks by the severity, so that we can develop a script that according to device connected to the USB, matches

the corresponding attack higher in the hierarchy.

An attack scenario and a proof of concept that fits the vulnerabilities researched are also given. This combination of attack scenario and proof of concept should also give raise to the awareness of the dangers associated with the USB port.

We also mitigate the exposed vulnerabilities with a special designed defense application. An application capable of preventing USB port attacks. In the general release of Android and in the vendor customization.

1.4 Outline

In the rest of the document, chapters are organized as:

Chapter 2 Details the Android security model and provides a view of the currents state of the vulnerabilities found on Android. It also details the Android attack surface. Lastly, we enumerate the most relevant exploits that are related with this work.

Chapter 3 Describes vulnerabilities that we have researched and used throughout the dissertation.

Chapter 4 Explains a proof of concept that makes usage of the vulnerabilities enumerated in chapter 3, as well as some results we had when exploring these vulnerabilities. We also present an application that is capable of mitigating USB port related attacks.

Chapter 5 Gives an overview of the work employed, explaining what was possible to achieve, also gives room for future work.

1.5 Publications

A. Pereira, M. E. Correia, and P. Brandão, “USB connection vulnerabilities on android smartphones: default and vendors’ customizations,” 5th Conference in the "Communications and Multimedia Security", 2014.

Chapter 2

State of art

In this chapter, we present the Android security model up to the current Android version, thus describing the Android attack surface. We also describe some of vulnerabilities and exploits found throughout the Android lifespan, which relates to our work.

2.1 Android Security Model

2.1.1 Linux Kernel

Android at its core uses the Linux kernel, which, from a security standpoint, is very advantageous. The Linux kernel provides a basic set of tools for the isolation and execution of processes and for secure Inter-Process communication (IPC) and other I/O functionalities, like the POSIX system permission [17].

Android borrowed tools for processes isolation from the Linux kernel and are very well tested. Such tools have been in the Linux environment for many years, which make a solid foundation that Android can rely on. Currently the last Android version, Android 4.4 KitKat relies on Linux kernel 3.8. The Android operating system is also a cut down version of the Linux kernel, which reduces the attack surface [18].

2.1.2 Android system permission model

When installing an application, Android uses functionality from Linux in which it attributes a unique user id over that application. Creating what is a called a sandbox,

isolating that application folder and files, as well as virtual memory and its IPC [18]. Applications with a unique user id are like different users in a Linux system. This way applications are restricted to only their own folders and files.

In Android as well as in Linux, there are privileged users, like root and system, these users have a static id. The system user is slightly less powerful than root. Root user exists for running critical system tasks and should not be available to the user. The system user is for less demanding tasks, such as the control over application's settings, when the user wants to enable Wi-Fi communication, for example.

Android has also added some features to the permission system in the Linux kernel for their own infrastructure [19]. For an application to have access to common resources of the smartphone, such as Internet, phone calls, read SMS. It needs to be in the correct group id, each one of those permissions has a unique group id associated with it. The developers have to declare which permissions they need in the AndroidManifest.xml file, which goes with the application. If the user accepts those demands, the application will be associated with that collection of group ids, granting access over all the permissions asked.

```
cat /proc/3213/status
Name:   android.youtube
State:  S (sleeping)
Tgid:   3213
Pid:    3213
PPid:   1326
TracerPid:      0
Uid:    10043   10043   10043   10043
Gid:    10043   10043   10043   10043
FDSize: 256
Groups: 1006 1015 3003
```

Figure 2.1: Status of youtube application

For example, in figure 2.1 it is possible to see that the youtube application is member of groups with ids 1006, 1015 and 3003. Therefore, youtube application has permission to access camera devices, external storage write access and Internet access. In figure 2.2 we can see an extract of the Android filesystem config file, in which those permissions are declared and the associated ids.

```
#define AID_RADIO         1001  /* telephony subsystem, RIL */
#define AID_BLUETOOTH     1002  /* bluetooth subsystem */
#define AID_GRAPHICS      1003  /* graphics devices */
#define AID_INPUT         1004  /* input devices */
#define AID_AUDIO         1005  /* audio devices */
#define AID_CAMERA        1006  /* camera devices */
#define AID_LOG           1007  /* log devices */
#define AID_COMPASS       1008  /* compass device */
#define AID_MOUNT         1009  /* mountd socket */
#define AID_WIFI          1010  /* wifi subsystem */
#define AID_ADB           1011  /* android debug bridge (adbd) */
#define AID_INSTALL       1012  /* group for installing packages */
#define AID_MEDIA         1013  /* mediaserver process */
#define AID_DHCP          1014  /* dhcp client */
#define AID_SDCARD_RW     1015  /* external storage write access */
#define AID_VPN           1016  /* vpn system */
#define AID_KEYSTORE      1017  /* keystore subsystem */
#define AID_USB           1018  /* USB devices */
```

Figure 2.2: Hardcoded permissions with their asssociated ids from [1]

2.1.3 ADB key pairing

An USB connection and an Android Debug Bridge (ADB) enabled Android, could pose a serious security threat. So serious that since Android version 4.2.2, Google made a security enhancement to the ADB connection. Making sure that every USB connection has an accepted RSA key pair to the host computer that the Android device is connected to. So every new USB host that the Android smartphone tries to connect to, has to have been previously accepted by the user [20].

2.1.4 Application Signing

Each application for Android must be self-signed [21], without any central authority. So the developer of any application has to digital sign it with its own private key, which is provided when creating a developer account.

Applications can request a shared id, from other applications, requesting that both applications share the same id. In that case, Android shared user mechanism will detect if the entity that signed the application is the same. Developers can request permissions granted to other applications, if they share the same signature.

2.2 Analysis to Android security

2.2.1 Patch cycles

When a new vulnerability is found a new patch must be developed and delivered as fast as it can be, so that it reduces the number of infected devices. However, this is not exactly what is happening.

As we mentioned, Google allows vendors and carriers to modify Android to satisfy their needs. Enabling vendors to customize the device with default applications and functionalities. This results in a timely increase on the patch release. First the patch has to pass on to the vendor and then to the carrier and only then they will reach the end user [2].

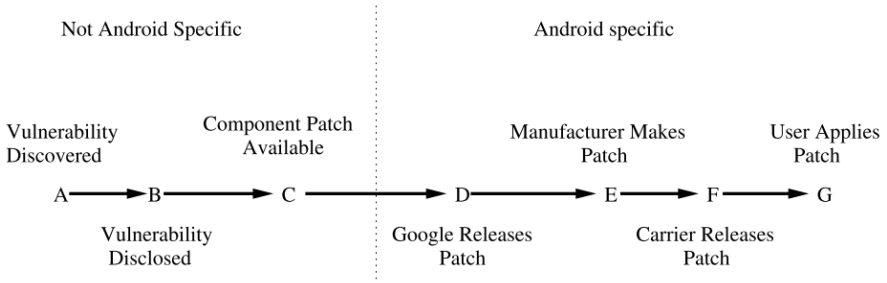


Figure 2.3: Life cycle from vulnerability found and a patch to that vulnerability be released in Android system permission file from [2]

As shown in the Figure 2.3, a vulnerability is discovered in step A, then disclosed in B, which is when Google starts developing a patch. Google releases it in step D, then two additional steps are required, until it finally reaches the end user. Step E. Step F is the step where the carrier releases it to the end users. Note that it is up to the manufacturer to release or not a patch to the users, sometimes it is not. It is estimated that the gap on steps E and F is of four months [2].

2.2.2 Common vulnerabilities and exposure analysis

Common Vulnerabilities and Exposures (CVE) accounts and catalogs known vulnerabilities of some technology or product, like Android. This information is taken from the

National Vulnerability Database[1]. When a vulnerability is disclosed, it is given a single identifier. The NVD catalogs this information, with a description, a threat level score and associated type of attack, like privilege escalation, memory corruption, etc.

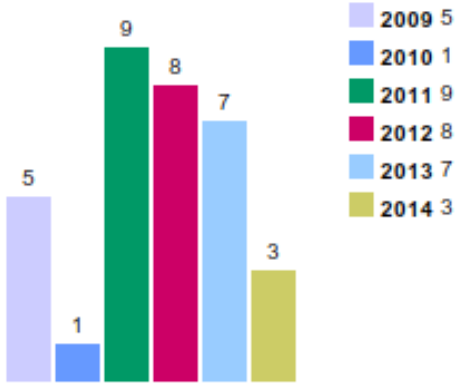


Figure 2.4: Number of Android vulnerabilities by year from [3]

Figure 2.4 shows the number of disclosed vulnerabilities by year, from 2009 until 2014. 2010 being the year with the lowest number of disclosed vulnerabilities. The year with the most disclosed vulnerabilities is 2011.

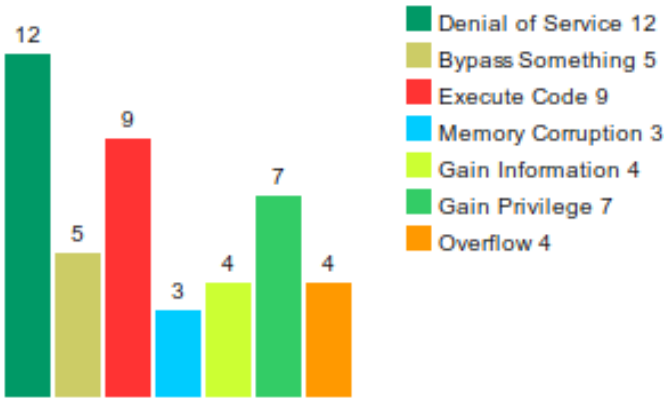


Figure 2.5: Android vulnerabilities by type from [3]

Figure 2.5 represents the number of vulnerabilities by type, with Denial of service having more vulnerabilities disclosed, followed by code execution. The types with less attached vulnerabilities are memory corruption attacks and gain information attacks.

Figure 2.6 presents the span of vulnerabilities according to the CVE threat level score,

[1]NVD is the U.S. government repository of standards based vulnerability management data http://nvd.nist.gov/

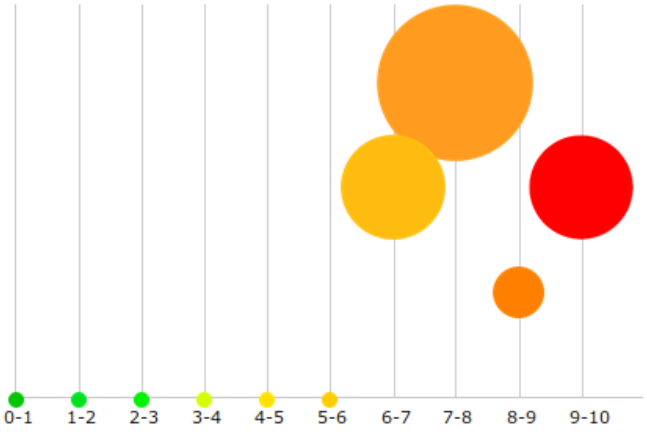


Figure 2.6: Android vulnerabilities by threat level score from [3]

where the highest score means that the vulnerability is more damaging. The size of the bubble represents the number of vulnerabilities disclosed, the color the threat level, which is also represented by the location of the bubble according to the y axis. It is possible to see that the vulnerabilities disclosed for Android are in general serious. This Information on the last figure 2.6 is from 24th of April of 2013 until 28th of April of 2014.

2.3 Android Attack Surface

Attack surfaces are logical paths where systems could be vulnerable. They provide a means to an end. Studying surfaces could lead to attack vectors, in a sense they are exposed areas in which the system could be vulnerable. Attack vectors on the other hand are more about what is, instead of what could be. An attack vector could be the use of a specific vulnerability from an attack surface. For example, an attack surface could be the mailing system of some network. An example of an attack vector could be a vulnerability inside the e-mail filter system. To that network, one of its attack surfaces would be the mailing system.

Since Android encompasses a large range of operations, from web browsing to SMS, games, applications, calls, GPS, etc. Its attack surface is of some magnitude. There are three main categories of attack surfaces. These are the remote attack surface, local attack surface and physical attack surface.

2.3.1 Remote Attack surface

The remote attack surface is where the attacker exploits vulnerabilities from remote access. Such as a direct attack to the smartphone's IP. It also could be adjacent, where the attacker is in the same network as the smartphone. Alternatively, it can also be by an intermediary. An attacker could fake a description and utilities of an application, make users download and install a malicious application from Google app store. Although, since February of 2012, Google launched a security service to the app store, called Bouncer, which scans the application for evidences of malware.

From a remote access, using an intermediary, the attacker could also explore vulnerabilities on the client side of the structure, such as the web browser, specially the webkit engine inside the smartphone. Attacking the webkit, could not infect just the web browser, but other web powered Application Programming Interface (API).

These kinds of attacks are likely to be the preference of the attacker. It would be the first step pursued to gain entrance on the system and further compromised it.

2.3.2 Local Attack surface

The local attack surface is where the attacker achieves local execution of code. In comparison with the remote surface, in which the attacker explores vulnerabilities of remote access, to achieve local execution of code. In the local attack surface, the attack explores vulnerabilities from inside the smartphone. This could be done through malicious applications, ADB shell. This way an attacker only has common access to the smartphone, the attacker is limited to that.

An attacker upon this stage would want to escalate the privileges from a common user, where he is in. This escalation will increase the power of access that the attacker has to execute code, explore the smartphone, as well as use the access to scan information about the user.

2.3.3 Physical Attack surface

With the physical attack surface, an attacker explores vulnerabilities from physical structures such as USB or NFC. This leads the attacker to use obstructive or non-obstructive attacks. In the first case, the attacker steals the device for a short time span, uses a portable device, with computational capabilities to inject malware on the device. In the latter case, the victim leads himself to being infected, like when he wants to charge the smartphone, it could lead him to connecting the USB to a public infected kiosk.

2.4 Attacks and Exploits

2.4.1 Injecting SMS Messages into Smart Phones for secure analysis

With the introduction of second generation cellphones, SMS has been used frequently for people to exchange information in relativity short time. By exploring the way in which SMSs are sent and delivered in smartphones, the authors of [4] developed a high-degree Man-in-the-Middle Attack (MITM) attack to the SMS infrastructure of the three main smartphone operating systems. Namely, iOS, Android and Windows phone. With the purpose of analyzing the secure vulnerabilities within the SMS infrastructure through fuzz testing. The attack outline is the same for all three operating systems, but the implementation is different for obvious reasons. In iOS all baseband communication is dealt by CommCenter process, which uses sixteen different serial lines as interface communication with the modem. The Android OS relies on the Radio Interface Layer (RIL), as common interface in the application framework for the application layer. The way in which the actual communication is done after the application framework, may vary from vendor to vendor, typically a single daemon communicates with the modem also through a serial device file. The windows mobile operating system is a bit different from the latter two, since it is larger and more distributed due to the fact that just for SMS communication, the application relies on a library just for that process.

We will focus on the Android implementation of the attack. The authors develop what they call an “injector” that stands between the telephony stack and the actual modem, intercepting the sent and receive AT commands, as it is possible to see in figure 2.7. This injector conceptually is the same for all the MITM attacks they use for every OS.

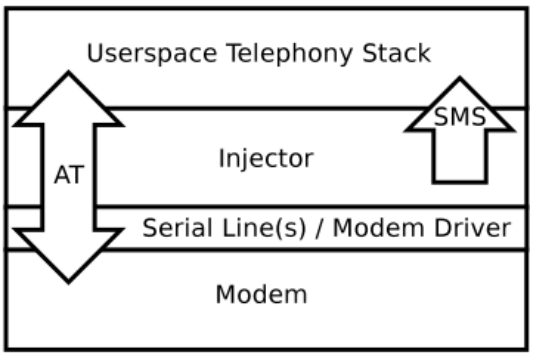


Figure 2.7: Conceptual model of injector from [4]

On the Android operating system, the implementation was done with a single daemon, this daemon connects with the serial line device whose purpose it is to talk with the modem. For the injector to work, First they rename `/dev/smd0` (the original device file for communication) to `/dev/smd0real`, then open the `/dev/smd0real` and then create a fake `/dev/smd0`. Upon restart of the Radio Interface Layer Daemon (RILD), it will communicate with the fake `/dev/smd0`, the modem will communicate with `/dev/smd0real`.

So the communication is, `RILD` ⇔ `/dev/smd0` ⇔ injector ⇔ `/dev/smd0real` ⇔ modem. This is so because the renamed original still holds the physical properties to talk with the modem.

With this, the injector has the ability to listen to every message sent and received, as well it has also the ability to send messages.

The injector listens to TCP port 4223 on all interfaces, to receive the SMSs that are about to be sent, by a computer on the same Wi-Fi network as the smartphone.

Fuzz testing is a kind of test, wherein the program being tested receives random structures as input, simultaneously being monitored for the kind of problems and bugs it might arise. The purpose of the authors is then, to implement this MITM to explore and find vulnerabilities on the SMSs infrastructure. A SMS message could be of several types, such as:

- **Basic SMS Messages**, typical SMS message.
- **Concatenated SMS Messages**, concatenated SMS messages.

- **Basic UDH Messages**, UDH stands for User data header, it is the binary structure presented at the start of a SMS, could convey different types of information, representing the type of SMS being sent.
- **UDHPort Scanning**, On the user data header, you could register the SMS for a specific port, SMS applications could listen to that port, similar to TCP port. With this, the authors scanned the ports, monitoring which applications were listening.

In the Android analysis, monitoring was done over ADB . That way it is possible to read the Android `logcat`.[2]

With that, it was possible to monitor the events and bugs that some SMS messages might induce. In the Android system, it was also possible to discover DOS attacks, namely the crashing of the com.Android.phone, which represents the Android phone application. Also the locking of the SIM card. Note that this is only possible if the actual SMS codification that causes the bug, is not modified or filtered over the GSM network, the authors were not specific about it if it were or not.

2.4.2 OldBoot

Oldboot is one of the first Android bootkits ever found in the wild. In 17th of January of 2014 it was estimated that it had infected more than five hundred thousand Android devices in China only [5]. There were no reports of infected devices outside of China. Which leads to believe that the attack is executed either by proximity, like USB connection or Wi-Fi network.

Oldboot affects the boot partition of the Android operating system injecting it with malicious content, making it undetectable by normal procedures.

Figure 2.8 shows how the infection is orchestrated:

1. Upon boot, the `init.rc` inside the boot will launch `imei_chk` as a system service and a socket attached to it.
2. The `imei_chk` will extract both the `libgooglekernel.so` to `/system/lib/`, the library folder and `GoogleKernel.apk` to `/system/app/` the applications

[2]Logcat is the main logging system, providing a way to report and log events, crashes etc.

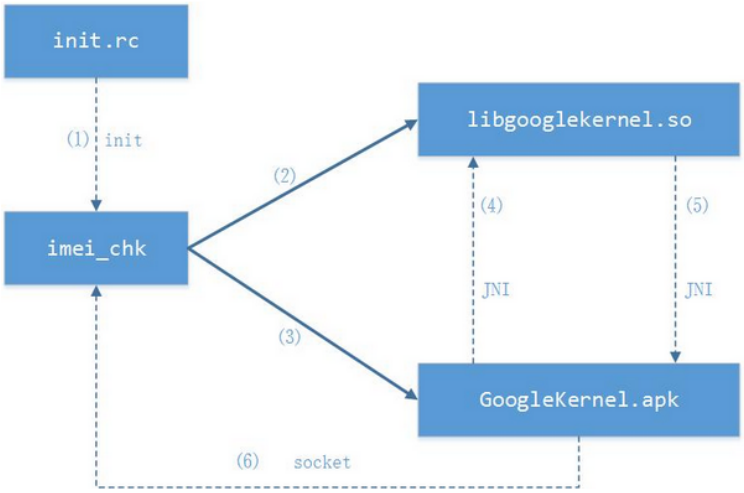


Figure 2.8: Oldboot file scheme from [5]

folder.

3. Since the `GoogleKernel.apk` is in `/system/app/` it will be installed as an application when booting. The application makes use of JNI, java native interface to communicate with the library `libgooglekernel.so`

4. The `libgooglekernel.so` being a .so file, a static compiled library, can be called by any process with permission to access it, to execute something, which is what `GoogleKernel.apk` does. Triggering malicious content.

5. The `GoogleKernel.apk` then sends shell commands to the `imei_chk` by the socket with port number 666, which was registered when booting, by the `init.rc` file, execute them. Since `imei_chk` is running as root every command will be executed as root.

If the device is infected, any application can use the same port 666 to send commands in order for the `imei_chk` to execute, putting the infected device in an even more dangerous position. The role `GoogleKernel.apk` application is to connect to command and control servers, in order to receive commands to be executed by the attackers and later executed by `libgooglekernel.so`,so `GoogleKernel.apk` is really a hidden complex Remote Access Tool (RAT).

The Chinese researches that discovered oldboot, also made available an application to detect and disable the malware.

2.4.3 Samsung Galaxy Back-door

A team of Samsung replicant developers [22], found a vulnerability in most Samsung Galaxy smartphones that enabled the modem operating system to access the internal storage of the Android operating system and write on it. This violates one vital security principal, the privilege separation. Typically, the modem should not have the ability to write inside Android, also Android should not have the ability to write inside the modem, since they are both separate things. Both should communicate by a standard protocols, whose features are well known, like the AT commands for the GSM configuration.

The problem is in the Samsung RIL, which handles communication from Android OS to the modem OS. It implements a feature that allows the modem to issue Remote File System (RFS) requests.

For a proof of concept, the developers had to compile a new modem kernel that could exploit this ability and install it in a modem used by a Samsung device. They were able to write a file in `/data/radio/test` with the string “hello world”. The default folder for writing is `/efs/root/`, however they could escape this using `../` to access the previous folder in the hierarchy, thus write anywhere. If the Android OS implements SE Linux, the scope of file to access may be restricted.

Serious threats could not arise, since first the attackers have to infect the modem first, but the Android OS should not have to rely on the security of the modem for its internal system, the privilege separation is a security principal not employed here.

The replicant developers have released a patch in their replicant software that mitigated the problem.

Chapter 3

Vulnerabilities assessment

In this chapter, we describe the vulnerabilities that we found on the Android operating system, namely the vulnerabilities found in its USB connection. We use USB connection with the Android smartphone as an attack surface, thus gaining entrance into the system.

Of those Android vulnerabilities, we give special emphasis to the vulnerabilities that we found on the vendor customization of the smartphone. Namely, we explore the vendor customization of the Samsung smartphone family. Some vulnerabilities are documented commands and others were discovered in our work.

We also describe an attack scenario capable of exploring the vulnerabilities described.

3.1 Attack scenario

In order for an attacker to exploit these vulnerabilities, it must control an USB connection to the victim device and hold on the other end of the connection a computational system capable of exploiting said vulnerabilities. An attacker could perhaps, hold a small computer with a USB connection and "borrow" the victims' device, for a short period, without him noticing it. In that, time the crux of the attack would be delivered, i.e. injecting the device with a compromised partition, which we will describe later.

A more practical scenario would be the installation of a public kiosk for charging devices' batteries, as the one shown in figure 3.1. Since the introduction of the smartphones, where the battery lifespan has been substantially reduced, compared to previous gen-

eration of cell smartphones, public charging kiosk, have become a more common scenario in public areas such as airports and libraries. This reduction in the battery lifespan increases the chance and consequently the number of times users find themselves without battery, which also increases the chance that the users would place their smartphones in said kiosk

Figure 3.1: Example of a public charging kiosk

Unknowing that the kiosk was infected, the victim on noticing that their smartphone has low battery, would try to find the kiosk as a quick fix to the problem and would willingly connect the device, hoping that it would charge the smartphone batteries. However, the true purpose of the kiosk would be to inject malicious code into the devices.

Such scenario is very fruitful, since we expect easy acceptance by the victim to place the smartphone in such a manner. There are a couple of reasons for this.

First, the lack of knowledge of the dangers attached with an exposed USB connection, given that is such an uncommon practice, even an IT experienced user could possibly lack this knowledge.

The second reason is the *emergency* state in which the victim is. Nowadays our cellphones are an extension of ourselves, it is completely implanted in our daily life and the lack of it is unthinkable. This is even truer for smartphones, since we can perform additional tasks on it, like smartphone banking and e-mails. Therefore, a state where the cellphone battery is empty or almost empty, would easily lead the victim to expose its device to the kiosk, in order to charge the battery.

Given the nature of such attack, a script is necessary on the computer holding the other end of the USB cable. A script capable of detecting the smartphone, match its vulnerabilities and proceed with the attack. For example, a different type of attack is executed for different Android versions, for different firmware versions, as well as different brands and different products of those brands. As an example, in the Samsung smartphone family, there could be an attack for the Galaxy S2 and another attack, using different vulnerabilities found for the Galaxy S4.

3.2 Vulnerabilities

The following vulnerabilities were researched and used throughout the implementation of the dissertation proposal. Some of them like ADB enabled, or the master-key vulnerability are known vulnerabilities, others like the ability to send proprietary AT commands over USB are not. It is important to note that neither ADB enabled nor the ability to send AT commands over USB, are by themselves a vulnerability, but having ADB enable as default, vendors' implementations of communication over USB with AT commands, make them so.

3.2.1 ADB enabled

The ADB serves as a means of communication between a computer (host) and a smartphone (device). The communication is done via USB, but it is also possible to configure the device so that the connection is made through Wi-Fi.

As default, ADB is disabled, but users tend to enabled it for various reasons, mainly among developers. ADB is enabled to develop applications and install them on the smartphone, to root the smartphone, for USB docking on the computer. Once enabled users tend to forget to disable it. According to a recent survey [23] 20% of users have ADB enabled, which is a significant number compared to the threat capacity that an ADB enabled poses as an attack vector. This number also indicates the lack of knowledge that users have of the USB threats.

With ADB enabled, it is possible to [24]:

- Get shell access to the device;
- Install applications that were not verified by the Google app store Bouncer;
- Uninstall applications;
- Mount the SD card;
- Read the `logcat`;
- Start an application;

Shell access through ADB could also unveil new attack vectors, as shown in [2], where it is possible to gain privileged access, with rooting techniques like Super One-Click root, also Cyndia impactor [25, 26]. The latter one explores a much more recent vulnerability affecting Android smartphones with versions prior to 4.2.

3.2.2 AT Commands

The AT Commands (ATC) are a command language that was initially developed to communicate with the Hayes Smartmodems, nowadays it stands as the standard language to communicate with some types of modems. For example, protocols like GSM and 3GPP [27] use this type of commands as a standard way of communicating with the

modem. With the ability to issue these commands to the modem, it is possible to:

- Issue calls;
- Send SMS;
- Obtain contacts stored inside the SIM card;
- Alter the PIN;

In order to understand this attack vector, we need to comprehend how a modern smartphone works. A modern smartphone is built with two operating systems, running in two very different environments [28]. On one hand, there is the Applications Processor (AP), where the Android operating system and all the applications with which the user interacts run. On the other, there is the Baseband Processor (BP), where the entire cellular (ex.: GSM) communication is done and where the modem lies. Issuing AT commands is not the only way to communicate with the modem, but it is the most popular way, together with Remote Procedural Calls (RPC).

The RIL, Radio Interface Layer [29], is the layer on the operating system responsible for establishing a communication between the Android operating system and the modem. In the case the application framework needs to send messages or make calls, it uses the RIL in the application framework to do so.

Figure 3.2 details the communication stack, from the application framework to the baseband. Where in the application framework, the RIL uses the Linux IP stack to communicate with the baseband, establishing the communication channel.

The RIL handles two types of communication: **solicited commands** and **unsolicited responses**. Figure 3.3 illustrates a solicited command sent from the application to the RIL and then to the baseband. An example of an **unsolicited response** would be receiving a call. The communication is initiated from the baseband to the application framework and then to the application responsible to handle such event.

As shown in 3.3, the main purpose of the RIL is to match every **solicited command**, or RIL call, in the application framework to its corresponding AT command and send it to the modem. Only the RIL issues the commands to the modem. In this case the **solicited command** is dialing a number. The RIL layer sends the corresponding AT command to the baseband. As soon as the baseband finishes dialing the number, it

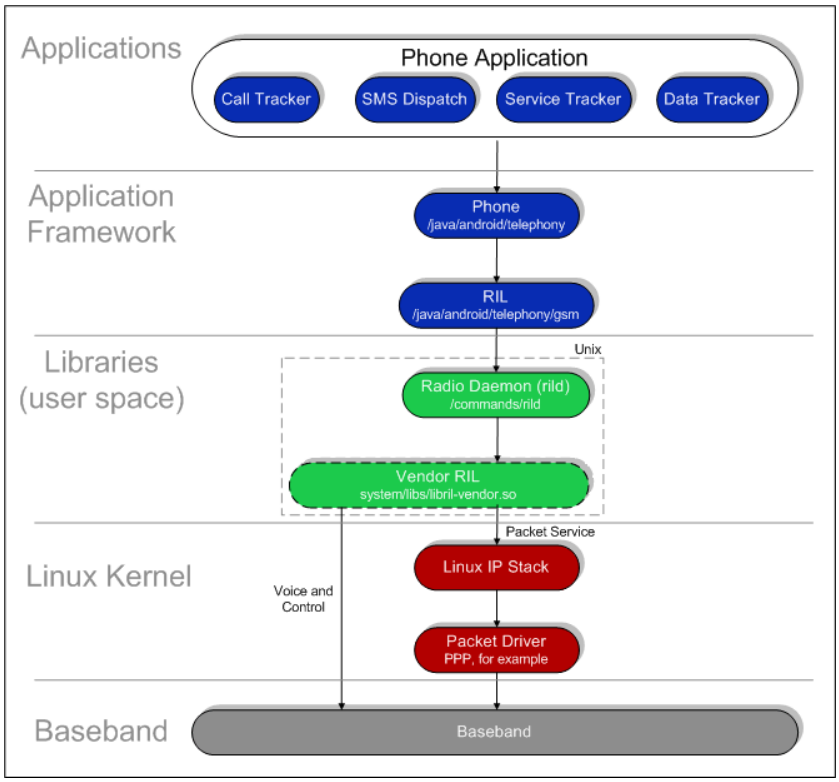


Figure 3.2: Android Telephony system architecture from [6]

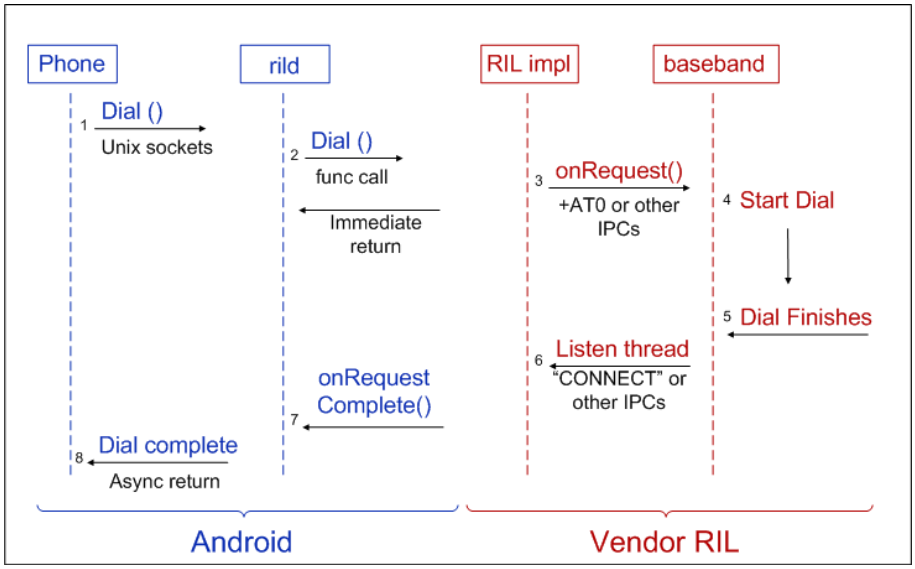


Figure 3.3: Diagram of a solicited call from [6]

notifies the RIL, which in turn asynchronously notifies the application that made the call, informing that the dialing has finished and providing with a socket to listen to the

call.

In this scenario, it is possible to use the USB connection to issue such commands. This vulnerability is only made possible due to the fact that some Android smartphone manufacturers, like Samsung and HTC, enable this through the USB channel. This means that it is possible to send AT commands using the USB connection, such as the `onRequest()` AT command from 3.3 directly to the baseband. It is important to mention that this is not a default feature of the Android, but a manufacturer addition.

In a practical scenario, upon the detection of a new USB device, the driver responsible for that device recognizes that the device has a modem type of interface through the USB, notifying the operating system of such. Henceforward the operating system has a way of communicating with the modem, which can be used in our attack scenario. Then it is conceivable to send AT commands to make smartphone calls or send SMS to premium cost numbers. This way, making the attacker earn money, or more accurately stealing it.

Complementing this, some manufacturers extend this list, adding some proprietary commands with the purpose of enabling control and capabilities that they want to have on the system. These commands are private and the manufacturers do not publish them. However, without the use of some form of encryption, anyone is able to eavesdrop the communication channel and understand or try to understand, what lies under the system.

3.2.2.1 AT Samsung proprietary commands

In the case of Samsung, a vast list of its family of smartphones has this vulnerability, where it is possible to communicate with the modem through the USB channel, without any previous configuration on the smartphone, something that does not happen with ADB.

More than the standard AT command set that comes with the 3GPP and GSM standards, Samsung extends this set adding their proprietary commands. This extends the capabilities of interaction that their computer Software (Kies) has on the device.

In fact, we discovered that the Kies for windows, uses both the standard set and the

extended proprietary AT command set, to at least achieve the following operations:

- Obtain contact book
- Obtain files in the SD card
- Update the firmware on the cellphone

3.2.3 Eavesdropping of Kies

Kies software was designed by Samsung to manage the smartphone over USB. Some of its functionalities include: the ability to update the smartphones' firmware, backup contact information, media exchange. We examined how this information was transmitted. The USB trace[1] eavesdropping tool was used to see what is transmitted over USB when a Samsung device is connected to PC and managed by Kies. Also, using Android `logcat`, we were able to see the nature of the messages that the Android internal processes leaked/logged.

With this approach, it was possible to observe something of interest, as pictured in figure 3.4. It shows that Kies uses a command system for the exchange of information.

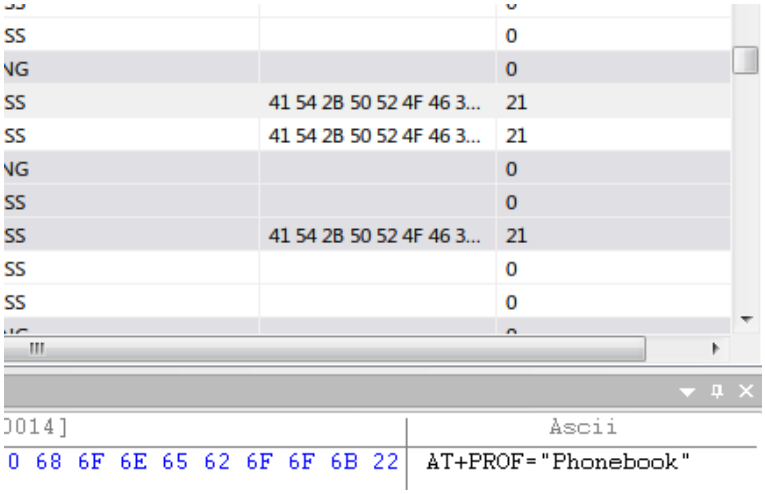


Figure 3.4: Eavesdropped AT command

Figure 3.5 shows the `logcat` of Android on Samsung S-5839i and we can see that there is a system process called `dun_mgr` that handles the communications from the USB to the `atx` process, `dun_mgr` obtains the commands transmitted over USB and

[1] USB Protocol Analyzer http://www.sysnucleus.com/

sends them to `atx`, which then executes and responds back to `dun_mgr`. The `atx` process logs its execution with an AT label when it is an AT command and with ATX label when it is an `atx` command. We assume from the log that the ATX label is for commands that are standard to the GSM 3GPP protocol and the AT label is used when logging Samsung proprietary commands.

```
dun_mgr    AT command: at+devconinfo
at         ATcmd:0 at+devconinfo 14
at         AT_ProcessCharParserDataInd at cmd sent state=4
dun_mgr    atx response: ...
```

Figure 3.5: Logcat of AT commands

The command `AT+DEVCONINFO` is the first command sent by Kies to the smartphone that we could eavesdrop. As shown in figure 3.6 it returns information about the smartphone, such as the model, the firmware version and the IMEI.

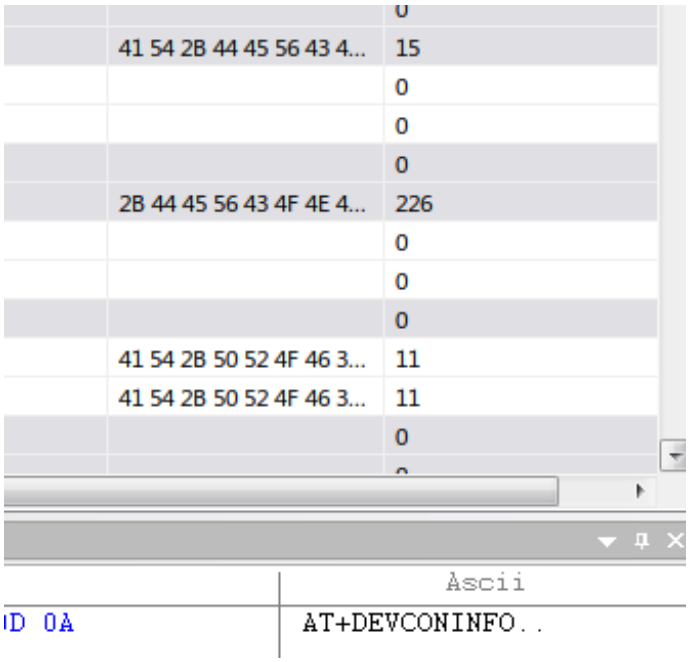


Figure 3.6: Eavesdropped of the command `AT+DEVCONINFO` being sent

After this initial exchange of information, the `atx` process on Android starts the `NPS_MOBEX` service, with the command `pcsync_write`, as shown in Figure 3.7. After that, we cannot make sense of the rest of the communication shown in figure 3.8.

```
atx                    pcsync_write
NPS_MOBEX              ################ OBEX size:35
NPS_MOBEX              nTotal = 35 nsizeobex =35
NPS_MOBEX              wscMessageGet msgid:4100
NPS_MOBEX              wscObexAdapterEventHandler param1:1
NPS_MOBEX              wscObexEngineGetWorkSpace
```

Figure 3.7: Start of the `NPS_MOBEX` service

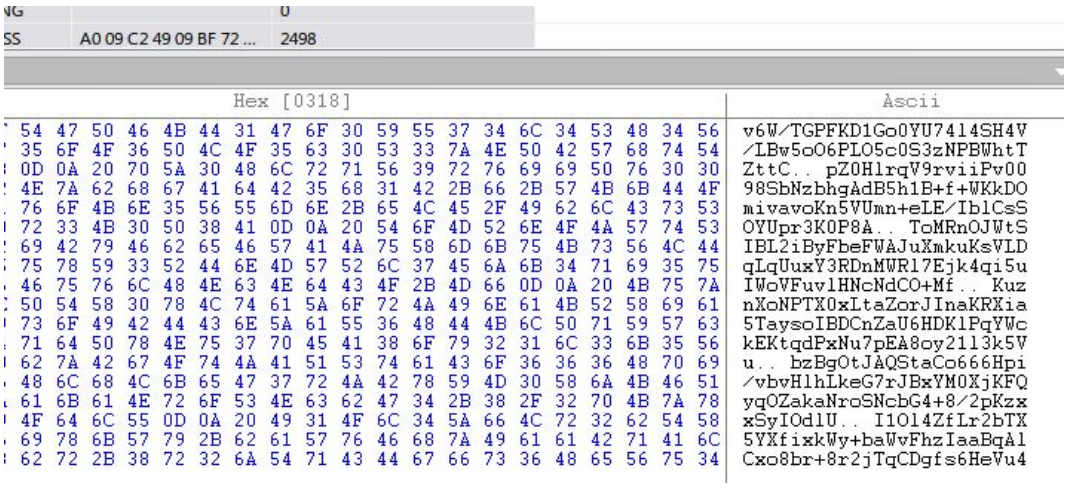


Figure 3.8: Eavesdropped information on a latter stage of the communication

3.2.3.1 Samsung internal processes

On Samsung S-5839i and using `lsof`[2] command tool, it is possible to investigate a bit further. In figure 3.9 shows that two different pipes connect both `dun_mgr` and `atx`. Another process, `usb_portd`, which purpose is still unknown, which it shares a socket connection with both `dun_mgr` and `atx`.

On Samsung GT-I5500 it is a different approach, in this case there is only one. Wherein the S-5839i there were two processes for handling communication, here there is only one. The process name is `drexe`, it was possible to extract some proprietary AT commands from the process file. Reading the ASCII values of file, the process leaks precious information about which AT commands are proprietary.

Figure 3.10 lists the AT commands successfully extracted, reading the ASCII values of

[2] `lsof` stands for "list open files" which is used in many Unix-like systems to report a list of all open files and the processes that opened them

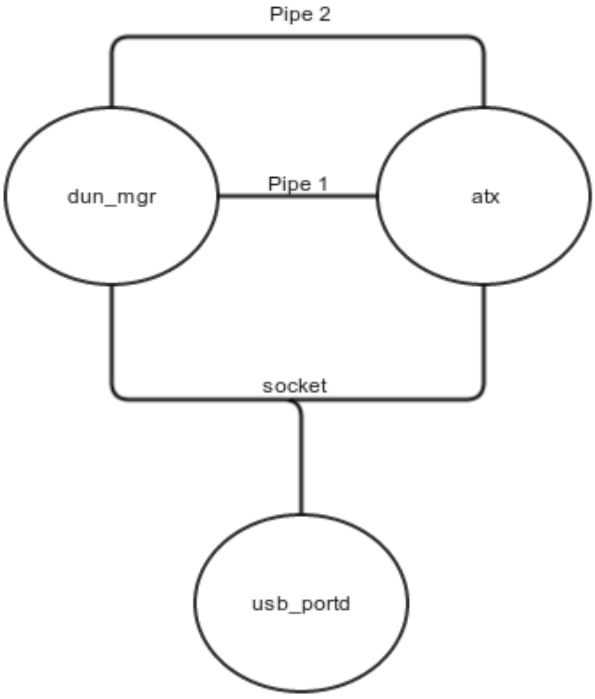


Figure 3.9: Samsung AT commands IPC configuration on s5839i

the process `drexe` from device GT-I5500. Several commands mentioned here, were also present in an online forum[3], upon experiment with these commands by issuing them through the channel and using once again realterm[4]. It was possible to note that most of them are useless for an attack and are specific to the version of the Samsung customization. Because it only works with the device GT-I5500. But by issuing the command on the third line, `AT+FUS?`, we managed to place the smartphone instantly in download mode in a vast array of Samsung devices listed on section 4.4.1.

The discovery of these commands reinforces the idea that, in fact there is some kind of filtering done by the operating system of the AT commands. Why Samsung uses this interface as a way to do non-baseband communication, such as this case, is beyond our comprehension. Since AT commands where made to communicate with the modem and not to place the smartphone in download mode.

[3]Forum XDA developers http://tinyurl.com/lekmpdv

[4]Realterm is a terminal program specially designed for capturing, controlling and debugging binary and other difficult data streams. http://realterm.sourceforge.net/

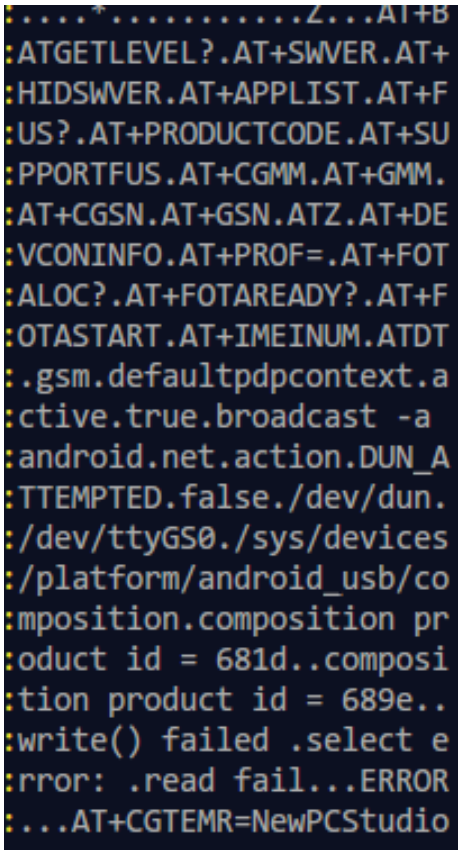


Figure 3.10: ASCII value of the process, where it is possible to extract a list of proprietary AT commands

3.2.3.2 Command "AT+DEVCONINFO"

The execution of the proprietary command `AT+DEVCONINFO`, in addition to obtaining all the information displayed in 3.11, also triggers the smartphone to mount the SD card on the computer as a mass storage device. As mentioned, when we eavesdropped the connection, it was possible to confirm that this was the first command sent to the smartphone by the Kies software. This assists the program in gathering all the necessary information in order to work properly. As it can be seen in figure 3.11, the information includes the smartphones' firmware version, the smartphone model and IMEI.

We also execute this command for identification of the device, since it gives all the important information needed for identifying of the correct partition versions of the device. For example in the figure, 3.11 shows that the device version, is S5839iBUlc1, then s5839iCLLB2, s5839iBUlc1 and s5839iBUlc1. This helps us further identify the

smartphone firmware, but we will further explain why we need this, in chapter 4.

Figure 3.11: Response message from issuing `AT+DEVCONINFO`

3.2.3.3 Command "AT+FUS?"

With the execution of the `AT+FUS?` the smartphone is placed in download mode, which is the mode where all Samsung smartphones need to be in order to flash their internal partitions and thus update the smartphone's firmware. Normally putting the smartphone in download mode involves mechanical key pressing, i.e., input from the user, which implies acceptance to flash the smartphone. This is not the case when it is triggered by the AT command, no user intervention is needed, which enables an automation of this process. The discovery of this command, led us to assume that this how Kies updates the firmware. Download mode is similar to fastboot mode that you find in other Android smartphones, but in download mode it is impossible to use the fastboot tool for flashing the firmware, which is included inside the Android SDK, only Odin and Heimdall serve that purpose, which will be discussed further at 4.1.

This is the most damaging of the vulnerabilities, since it is possible to alter the partitions inside the smartphone, explicitly injecting code on the desired partition. The novelty of this attack, like the `AT+DEVCONINFO` command, is that no prior configuration is needed on the smartphone. This is achievable just right after the smartphone is bought. Placing the smartphone in download mode enables us to use Odin to flash the smartphone with malicious partitions.

3.2.4 Master-Key vulnerability

Disclosed at Black Hat USA 2013 by Jeff Forristal [30], the master-key vulnerability affects devices from Android 1.6 Donut through 4.2 Jelly Bean. Here the package manager does not properly verify the signature of an application. Making it possible

to change an application execution code and still hold a valid signature of original application.

An `.apk`, which is the Android application file, is nothing more than `.jar` file with more features, a `.jar` file is it is like a `.zip` file. In a `.apk` and in a `.jar` file there is a file called `MANIFEST.MF`, which contains all the hashes of every file in the `.apk` or `.jar` file. When you use jar signer on both of the types, an additional file is added, the `.SF` file, this file holds a signed verification of every hash on the `MANIFEST.MF` file. On top of that, in the `.apk` there is another file `.RSA`, which contains a signed hash of the `.SF` file and a public certificate of that signature. This way an `.apk` has the capability to hold integrity and to hold a signature and a public certificate of whom has signed it, even though, there is no central authority to the certificate, it is the responsibility of the user to trust that certificate.

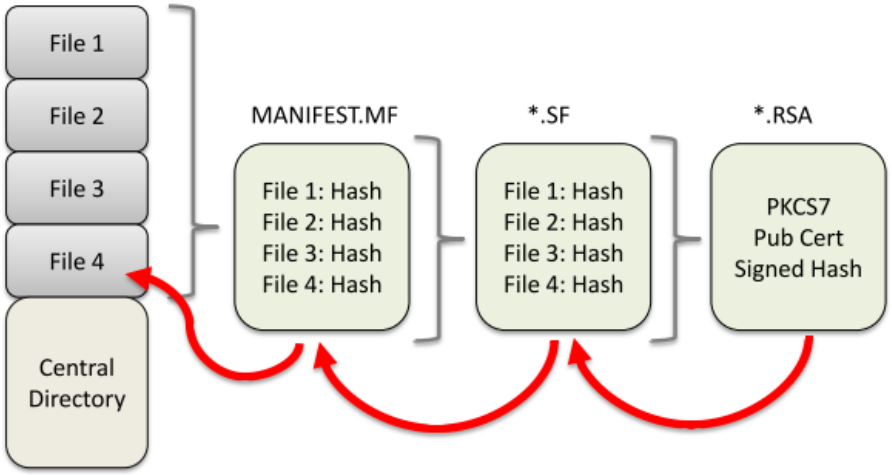


Figure 3.12: APK file structure

As a normal `.zip` file, as well as an `.apk` file, it can hold several different files with the same name, which is where the main problem lies. When that happens, the Android system or better the package manager, which is the process that installs each application, does not correctly verify the application signatures. This is the result of having two different kinds of hash list implementations for indexing files inside the `.apk`, one for signature verification, another is for loading content to actually run the application.

During installation the java implementation of the `LinkedHashMap` is used to verify the

signatures, this implies that when having two files with the same name, it will only store the last one found by name. Upon execution, the Davlik virtual machine, using `dexopt` written in C, uses and hashlist implementation written in C. What that does is that it will execute the contents provided by the first file indexed, so when having two files with the same name, it will index only the first one encountered.

So if there is a `.apk` file with two files with the same name, the first being the not signed one, the one that you want to run and the second being the original signed one, the one that you want to be verified. Android will verify the second and execute the first.

This has a tremendous implication on the security of the Android system, because you can alter an application code without the Android system knowing. Applications by signed by vendors have a special ability: they can ask for system permission. Using this vulnerability, it is possible to alter execution code of one application that has been signed by the vendor and with that gain system permission.

System permission is close to root and there are numerous ways to escalate from system to root, but not applicable to every smartphone due different Android versions that the smartphone may be running and security mechanism applied by the smartphone manufacturer.

Chapter 4

Implementation

In this chapter, we use the vulnerabilities found and described in the previous chapter, to explain in detail the attack process. That attack process is constituted by an attack scenario and a proof of concept that fits the attack scenario. With this, we detail the main architecture of the proof of concept, the different types of attacks used according to which vulnerability, as well as a general explanation of why it is possible to do so.

We provide a list of vulnerable devices regarding one type of attack. In addition, we develop an application for Android capable of mitigating the developed attacks.

4.1 Architecture

The script for the proof of concept has to be fast, fully automated, effective and able to perform operations on numerous levels of the OS stack.

As shown in Figure 4.1, we deploy a script in a guest virtual machine containing Xubuntu that, when necessary, is able to communicate with its Windows 7 host machine. We used a Xubuntu virtual machine so that the script can take advantage of the Linux OS scripting environment. Linux comes with libusb[1] as a generic driver to handle USB devices, which in turn lets us use the usbutils, thus making it more practical for scripts like this to be developed. We detailed its functionalities further down.

As virtualization software, we use Virtual Box[2]. This program enables the creation of

[1] C library that gives applications easy access to USB devices http://www.libusb.org/
[2] VirtualBox virtualization software https://www.virtualbox.org/

guest virtual machines and at running time it is possible to exchange the control of the USB device from guest to host and vice-versa. In order to give the guest control over the USB device, it presents a virtual USB controller to the guest, as soon as the guest starts using the USB virtual device, the USB device will appear as unavailable to the host. So only one OS has access to, thus can control, the USB connection at a time.

It is essential that the guest machine is Linux and the host is Windows and not the other way around. This guarantees that the Samsung device drivers have direct access to the USB device, which would not work inside a guest machine, because VirtualBox emulates the devices. The type of attacks done by the guest machine supports this emulation.

We have a host with Windows 7 so that it can have access to Odin[3], a tool that is used to flash partitions in Samsung devices. This tool is unable to work in Linux, because of the dependence that it has on the Windows Samsung drivers. Other tools are able to flash firmware on Samsung phones on Linux, like Heimdall[4], but it are only able to target a limited number of Samsung smartphones, whereas in the case of Odin it was possible to target all Samsung smartphones.

A communication channel between the host and the guest is needed, in order that the guest, which is doing most of the work can tell the host, when and which smartphone to flash. For that we used a shared file between the guest and the host, so when the guest needs to communicate it writes to the file. The host is polling the file for changes. The roles are exchanged when the communication is in the opposite direction. Another way to do this would be to hold a socket of communication between the two virtual machines, by creating a specified TCP port.

In the guest machine, it is necessary that we configure a full list of the USB devices that will be attacked, in order to tell the guest machine which devices to filter, so that they could be controlled inside the guest and not on the host. This is done by identifying the product ID and the vendor id, which together identifies all USB devices. The vendor ID identifies the brand, for example Sony, the product ID identifies the product, for example Xperia.

[3]Popular flashing tool for Windows http://odindownload.com/
[4]Cross-platform flashing tool http://glassechidna.com.au/heimdall

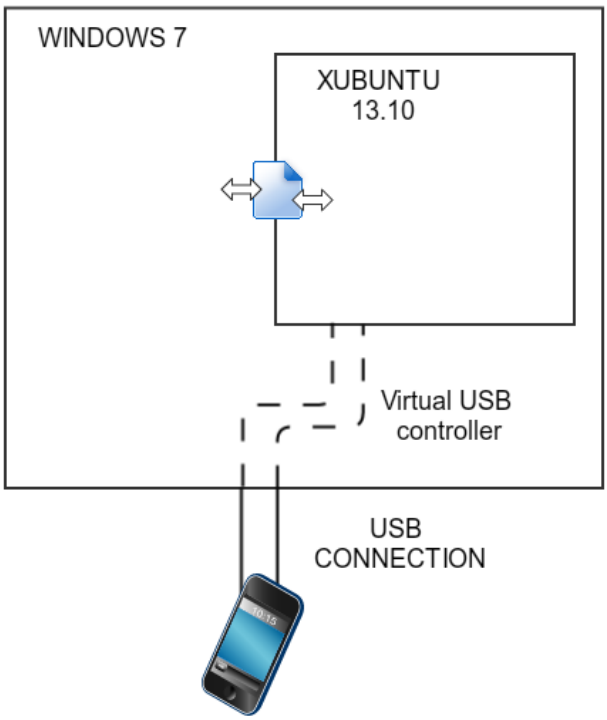


Figure 4.1: Architecture of communication Guest OS and host OS.

The script running on the guest is responsible for:

- Detecting plugged USB devices;
- Identifying the type of device;
- Matching of the vulnerabilities found on that device;
- Attacking using the known vulnerabilities;
- Communicating with the host, in case the vulnerabilities require the use of the Odin tool;
- Identifying the mounted external cards of USB devices.

The Windows 7 host is responsible for:

- Communicating with the guest, to know which device to flash;

- Identifying the flash image that matches the device and its firmware;
- Identifying the correct version of Odin for flashing;
- Use of GUI automation libraries, like Pywinauto[5], in order to use Odin without human intervention.

4.1.1 Control-flow of the attack

The algorithm running inside the guest Linux, aims to infect the plugged device with the maximum dose available, according to the vulnerabilities documented in section 3.2 that match said device. Different types of vulnerabilities lead to different types of attack.

We established a hierarchy of attacks and the main aim is to obtain an attack of higher order according to the vulnerabilities catalogued for the specific device. We call this a hierarchy, since all attacks of higher order in our assumption, encompass all the possibilities and more of each attack of lower order.

For example, the ability of flashing a compromised boot partition is of higher order than to just of have an escalated ADB attack, since in the flashing attack, the attack is more difficult to erase. This also is true for a privileged ADB attack, over a non-privileged attack, for example a non-privileged ADB attack to just have GSM AT command interface.

The following attacks are ordered using the hierarchy:

1. Having `AT+FUS?` command possibility
2. Having ADB with possibility to privilege escalation
3. Having ADB with no possibility to privilege escalation
4. Having GSM AT command interface

As shown in figure 4.2, we developed a script in order to follow that hierarchy and detect the attack of higher order to proceed.

[5]Windows GUI automation using Python https://code.google.com/p/pywinauto/

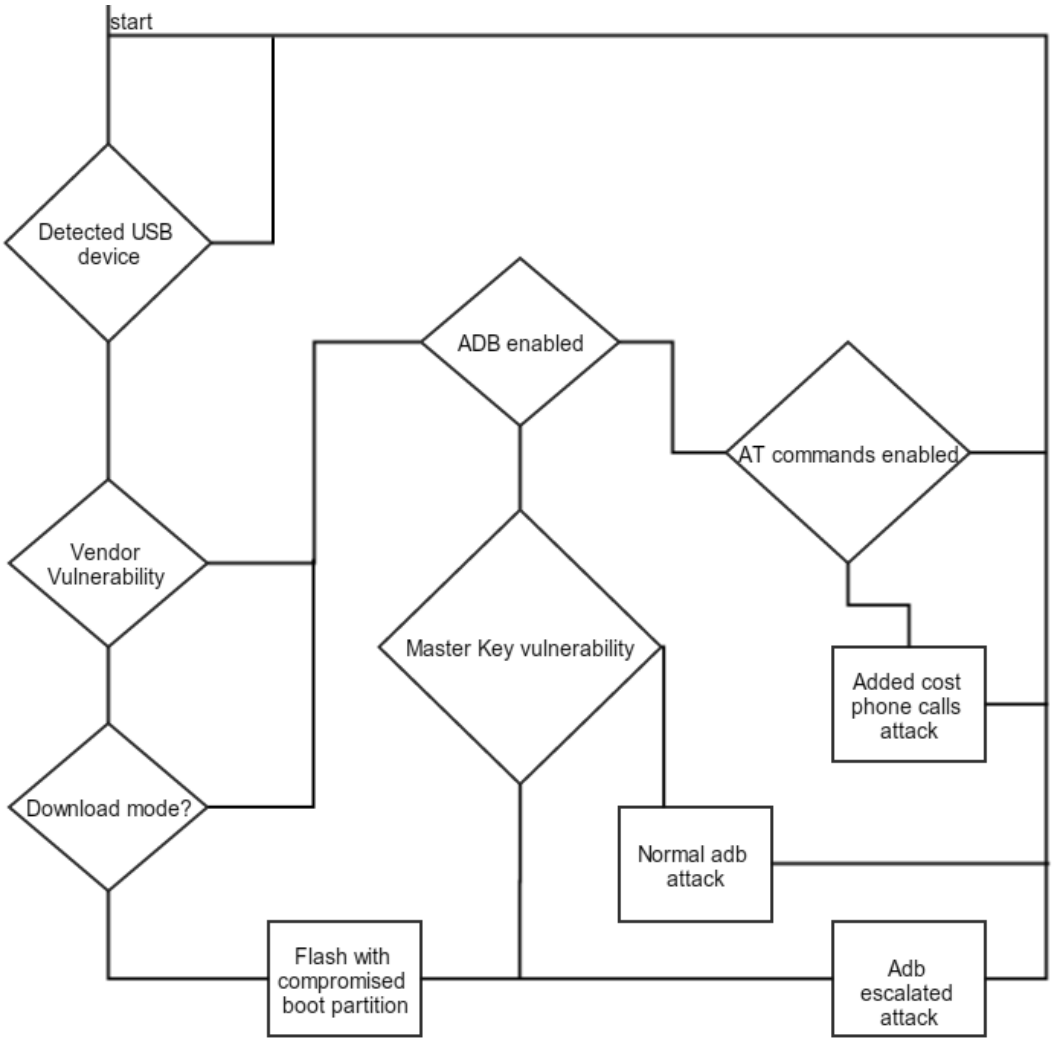


Figure 4.2: A Flowchart of the algorithm, detailing the implementation of the hiearchy

4.2 Using the vulnerabilities found

As mentioned in the previous sub-section, the guest machine detects plugged devices, identifies the smartphone version and platform, matches them to the vulnerabilities found and executes the attacks that target those vulnerabilities. In this section, we will cover the attacks.

4.2.1 AT command interface

Having the ability to issue AT commands to the modem, thus having the possibility to make calls, the attacker could for obvious reasons exploit this breach, to either make

calls to added cost numbers or, SMS subscriptions to added cost numbers. That way the attack has a monetary end. For that purpose it we used the following AT commands combination, when we have the possibility to issue AT commands over USB.

- **For SMS:**

```
AT+CMGF=1
AT+CMGS=+<ADDED_SMS_COST_NUMBER>
<SMS_TEXT>
```

- **For making calls:**

```
ATD + <ADDED_COST_CALL_NUMBER>
```

4.2.2 Enabled ADB connection

As mentioned, having ADB connection enabled could pose a serious threat. First, there is the threat that an attacker uses this vector for privilege escalation. Here the attacker explores the capability to escalate the privilege of simple shell access, to root or system. Then there is another possibility given by ADB enabled, of installing applications without user consent, for example, the attacker could install a surveillance application with maximum permissions available for an application. So a non-privileged ADB attack could do anything that explores the capabilities mentioned. It is really up to the attacker to choose which one. In our proof of concept, we use this vulnerability either to escalate privileges, or to install a surveillance application.

4.2.2.1 Privileged escalation and master key vulnerability

For privileged escalation, we chose the master key vulnerability, described in section 3.2.4, to gain system access. We picked the master key vulnerability, since it is one of the most recent vulnerabilities disclosed that enables privileged escalation. It affects a very wide range of Android versions, from 1.6 up to 4.2. Additionally the master key vulnerability is a flaw on the Android system, not just in vendor customization, thus it is possible to treat all Android smartphones alike.

Applications that are platform signed can request for system permissions on the mani-

fest file. Using the master key vulnerability, it is possible to change the execution code of a platform signed application, without any problems on the installation and execution.

For this use we chose an application signed by Samsung, in this case we choose a cisco Virtual Private Network (VPN) application, this application installation requests system permission in order to run, making it a good candidate for a malicious modification and installation.

We just need to compile a manifest compatible `classex.dex` file, the main calling class has to maintain the same name, but the actual implementation is changed to run any wanted code.

We used `7zip`[6] to alter the original `.apk` file, to add the malicious `classes.dex` in first and the original `classex.dex` second inside the `.apk` file. Doing it the other way around the vulnerability would not work.

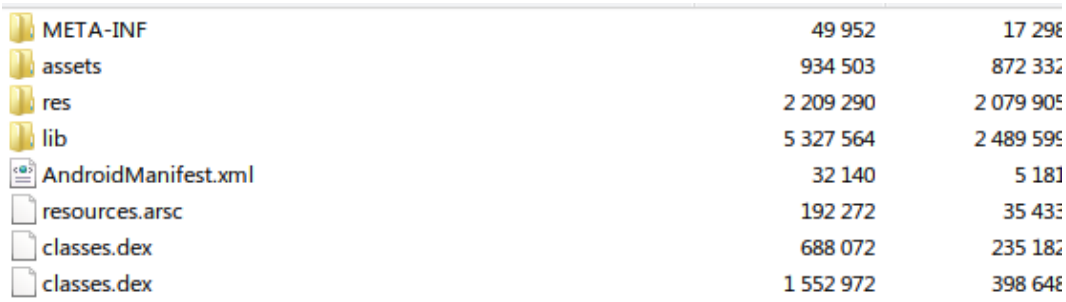


Figure 4.3: Files inside `.apk` after added compromised `classes.dex`

After having the malicious application created, we make use of the enabled ADB to install the malicious application. That is possible without any user consent, unless the Android version of the phone is above the 4.2.2 version, in which case the user must consent the ADB connection to the computer.

The goal here is not yet defined, but upon having an application with system permission on the phone, the possibilities are open. Depending on the Android version and the phone model, it is possible to escalate it further to root. In some versions of Android prior to 4.1, with system permission it is possible to change the local prop file. Re-writing the attribute `ro.kernel.qemu` to 1, this way, the operating system thinks it is

[6] 7-Zip is a open source file archive with a high compression ratio. http://www.7-zip.org/

an emulator instead of a real phone. This gives root access to the ADB connection and with this it is possible to root the phone.

In addition, it has full access to the data folder on the Android phone, which contains user data, contacts, messages and settings. This folder holds important information about the user and upon having system permission with malicious intent, it should not be considered safe anymore.

4.3 "AT+FUS?" and flashing of the boot partition

4.3.1 Device identification.

The purpose of this attack, is first to identify the firmware version of the smartphone with the command `AT+DEVCONINFO`, as it was shown in figure 3.12. This enables to identify the firmware version of the smartphone in question. For example in figure 3.12 it is possible to identify by `VER(S5839iBULE2/EUR_CSC/S5839iBULC1/ S5839iBULE2)`, which shows the following details as per the format **PDA, CSC, Modem and Boot**:

- **PDA**: The build version, which includes the system partition.
- **Country Sales Code (CSC)**: (Country Sales Code): Language and country parameters.
- **Modem**: Identifying the modem firmware version.
- **Boot**: For the version of the Boot partition.

4.3.2 Changing the Boot Image.

As described in section 3.2.3.3 the AT command `AT+FUS?` places the phone in download mode and allows flashing a new boot partition. The boot partition is where the first line of code to be executed, when any Android smartphone is booted, is. The primary functions of the boot partition are:

- to mount the primary partitions necessary for the system to boot;
- start the first process of all operating systems based on Linux, i.e. the `init` process;

- read and execute the `init.rc` boot configuration file;
- load the kernel and the ramdisk.

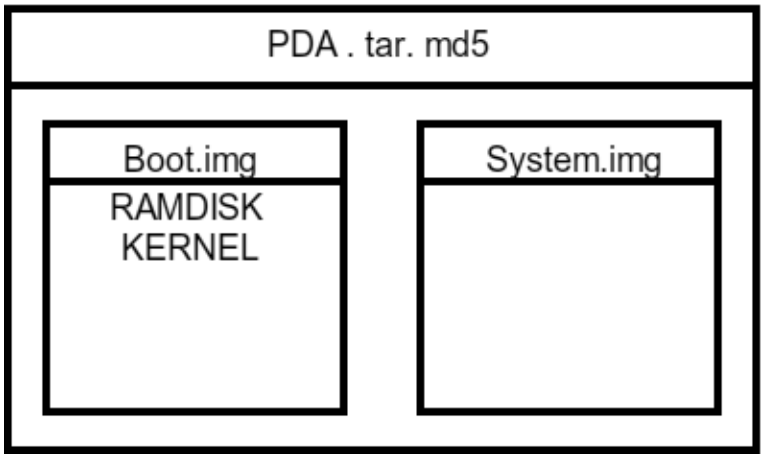


Figure 4.4: Structure of a typical PDA file.

Figure 4.4 illustrates a firmware file about to be flashed on a device. The boot partition is placed in a `boot.img` type file, which is inside of the PDA file. This file in turn is as tar file with a checksum, usually a .tar.md5. Inside the PDA file there is also the System partition. In order to flash we only need the `boot.img` file. For the proof of concept, we re-construct a stock boot partition.

The bootloader is not exactly the same as the boot image [31]. The bootloader is the very first piece of software that is executed on a device. It detects which boot images to initialize, in typical Android operating system the bootloader either starts the recovery mode or the normal mode. In most Android smartphones the boootloader is locked, which means that it verifies the integrity and signatures of the partitions before it starts them. This prevents any modification of the partitions, such as the boot or the system by an unauthorized third party. Samsung devices bootloader is typically unlocked, which means a further modification of the partitions is possible, thus enabling our attack.

Figure 4.5 shows in a very simple way the role of the bootloader, presenting one of its main functions, which is to boot either the recovery mode, or the normal mode, each explaining its functionalities.

In order to modify the `boot.img` file, it is necessary to extract the ramdisk and the kernel first, from the `boot.img`. Then modify what is needed and pack everything

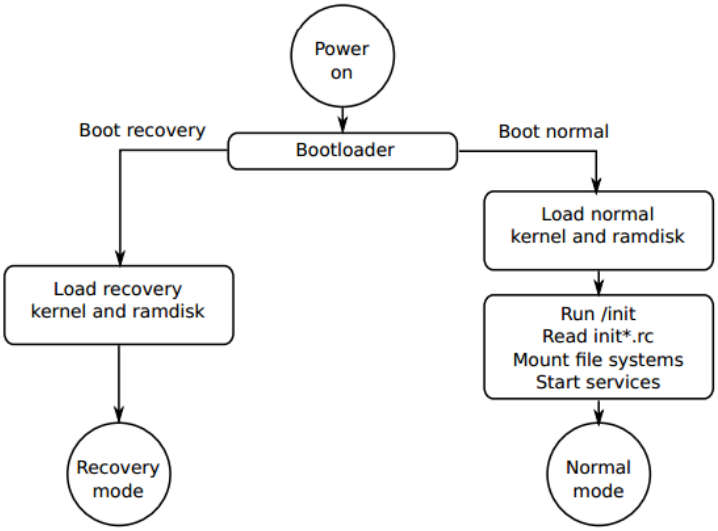


Figure 4.5: Boot sequence from [7]

again, in the correct file format. Normally the `boot.img` file is divided in two parts, the Kernel and the ramdisk. The ramdisk is the first file system mounted on the root of the system. There it is possible to find files like the `init` file and the `init.rc` [32], the image that is used in the booting process of the smartphone and various other folders needed for good performance of the system. In the `init.rc` file several shell type instructions are found for the initial configuration of the system, as well as instructions that will be executed with root access by the `init` process. It is in this file that the ramdisk of the boot partition will be altered, adding malicious instructions.

In this scenario we want to achieve three different goals by altering the boot partition, **first** we want to make the ADB always enabled, **second** we want to obtain root access, **third** we want install a surveillance application, in this case Androrat, which is described below.

4.3.2.1 First objective, make adb always enabled

For the first objective, we altered `on property:persist.service.adb.enable=0`. This property tells the system, what operations to do when disabling ADB. In the original file it stated that when it was to disable adb, it would stop Android Debug Bridge Daemon (ADBD), effectively stopping the ADBD on the smartphone. This was

changed from stop ADBD to start ADBD, rendering it impossible to disable the ADB from configuration, even thought it might appear disabled in the systems' options.

4.3.2.2 Second objective, obtain root access

For the second objective, we want to obtain root access on the smartphone using the `su` binary file [33]. This binary configured with permissions of execution for everyone and with the owner as root, enables any user to have root access. We placed the `su` inside the ramdisk, at boot it will be placed by the bootloader on the root of the system. Then we added the following lines in the `init.rc` so that it is copied to `/system/xbin/` folder.

- `copy /su /system/xbin/su`
- `chmod 06755 /system/xbin/su`
- `chown root /system/xbin/su`

This will allow root access to any user, for example when ADB is enabled it will have root access.

4.3.2.3 Third objective, install an surveillance application

For the third objective, we will install a surveillance application on the device, in this case Androrat which stands for Android RAT [34], in a way that the user does not know of its existence. First application .apk file is placed inside the ramdisk directory, so that once it boots, it places the .apk in the root of the system, similar to what had been done for `su`. And again the following code was added to the `initi.rc` file, so that it copies the file to the apps folder.

- `copy /AndroRat.apk /system/apps/AndroRat.apk`

Once it boots, the application is installed as a system app. This way the application can never be removed, even with root access, since it is permanently flashed on the boot partition and every-time the system boots, the application is installed.

The AndroRat application enables several remote surveillance capabilities on the phone:

- Get contacts (and all their information);
- Get call logs;
- Get all messages;
- Location by GPS or Network;
- Monitor received messages live;
- Monitor phone state live (call received, call sent, call missed);
- Take a picture with the camera;
- Stream sound from microphone (or other sources);
- Stream video;
- Do a toast;
- Send a text message;
- Make a call;
- Open an URL in the default browser;
- Vibrate the phone.

AndroRat is composed of a client and a server, the client is the application that is installed on the phone, the server runs on a PC, regardless if it is a Windows, Linux or Mac, since it runs in a Java virtual machine. The client communicates with the server by TCP/IP. In the original application the client launches an activity so that the users inserts the IP of the remote server, this has been altered so that the application can get the server IP by parsing a file on Dropbox with a static URL, thus not needing the user input, which makes the application unnoticeable. Also the original application `AndroidManifest.xml` file was changed, deleting the following lines:

```
<intent-filter>
   <action Android:name="Android.intent.action.MAIN"/>
   <category
      Android:name="Android.intent.category.LAUNCHER"/>
</intent-filter>
```

This way making the icon not visible in the app section of the smartphone, so an inexperienced user would not detect the malicious application.

4.3.3 Installing the new boot image

The boot partition is extremely sensitive to the smartphones' hardware, as well as the firmware. Every different type of hardware for the smartphone has its own specialized boot partition. That is one of the reasons that there is a vast selection of ROMS, to cover all the variety of hardware in the Android family.

For a successful attack, we created a dictionary of previously altered boot partitions, in order to encompass all the different smartphones in the attack list. For example, for the bundle version presented with `VER(S5839iBULE2/EUR_CSC/S5839iBULC1/S5839iBULE2)`, we map this bundle version to a previously altered boot image that matches this version.

After an initial identification of the smartphone version from the Linux guest machine, we place the cellphone in download mode, we also notify the host machine, so that it could hand over the control of the USB device to the host. When putting the device in download mode, its product ID and vendor ID is altered, to a non-filtered combination of vendor ID and product ID by VirtualBox, rendering the process of handing over the control of the USB device automatic. The guest machine saves the IMEI of the smartphone, so that once the phone reboots, it already knows that it is a compromised smartphone, which would enable the guest to do other type of attack, since the ADB is on and the phone is rooted.

After the host is notified to flash the device with the firmware version, it maps the given version to a previously altered boot image that is ready to be flashed. It also maps the correct version of Odin.

Using a GUI automation tool for Windows, the host commands Odin to choose the correct image file folder and name and then to flash the phone. The smartphone proceeds with the flashing of the partitions and reboots normally. After rebooting, its product ID and vendor ID changes once more to the previous ones, handing over again

the control of the USB connection to the guest machine, so that it is possible to proceed with the attack.

First the **Guest** Linux does:

1. Detection of plugged USB devices.
2. Matching of its vulnerability.
3. Checks if it has a compromised boot partition.
4. Notifies the boot image to flash the device and saves the IMEI.
5. Places the phone in download mode.

Then the **Host** Windows does:

6. Identification of which file matches give version.
7. Makes use of GUI automation tools to control Odin and flash the phone

And again the **Guest** finishes with:

8. Proceeds with the rest of the attack, now that it possesses root access and ADB enabled.

4.3.4 GUI automation tools

To fit our attack scenario, so that no human interaction is needed, we make usage of GUI automation tools, in this case Pywinauto [35], it is important that we mention these tools used and how we used them.

Odin is the reason why we need GUI automation tools, it is needed for the step responsible for flashing a boot partition. When issuing the command `AT+FUS?`, the device enters download mode, a series of GUI automation steps are needed for interaction with Odin, namely for selection of the boot image and also for clicking on the start button, to start flashing.

In the case of Samsung devices, Odin detects automatically when a device is ready for flashing, their USB product ID is altered internally when the device transits to download mode, in this way Odin detects that the device is ready for being flashed.

So we only need to the automation for selecting the Odin window, then select the boot partition from the folders and then to click in start download.

As shown in figure 4.6, Odin detects that it has a ready device for flashing, it is at this precise moment, which our script will start as well as its automation tool.

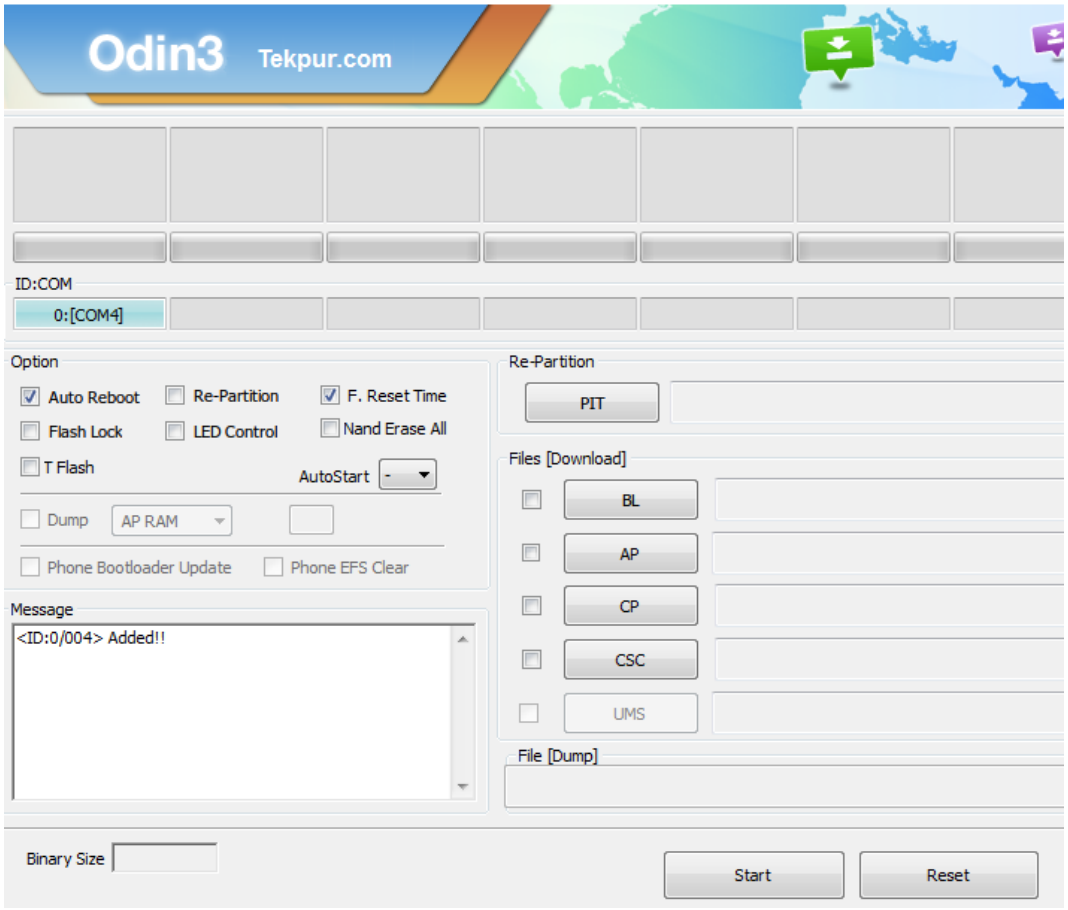


Figure 4.6: Odin program interface

First step of the automation tool, is to select the Odin window, so the program had to been previously launched.

The Ubuntu script, which handles the identification, sends the identification information of the device and the scripts running on Windows 7 will match the device identification to the correct folder and file of the boot partition to flash. We have a dictionary in Python, which matches the device information, with the correct boot file and folder.

After the automation tool selects the file, it just needs to click start in order that the device receives the partition file.

4.4 Tests

We performed several tests, all regarding the vulnerability that enables us to flash device, i.e., sending the command `AT+FUS?` over USB and successfully flashing the device with a compromised partition. First, we give a list of Samsung devices tested and a list of devices where the vulnerability was present.. After that, we give an analysis of behavior facing that vulnerability with certain popular anti-virus apps.

4.4.1 Tested smartphones

We successfully verified the attack on the following phones:

- Samsung GT-S5839i
- Samsung GT-I5500

And, by issuing the AT commands, it was possible to confirm that the vulnerability was present on the following phones:

- Samsung I9300 Galaxy S III
- Samsung GT-S7500
- Samsung GT-S5830
- Samsung I9100
- Samsung S7560M

It is necessary that the smartphone has an original ROM from Samsung. We expect that the span of vulnerable versions of Samsung smartphones be much more wide than this, since in our assumption, having the vulnerability or not is implicitly related with the ability that the smartphone has on communicating with Kies software. So as far as we now, most (if not all) Samsung smartphones are supported by Kies.

4.4.2 Tested Anti-Virus

AVG, **Avast**, **CM Security** and **virus scanner** [36, 37, 38, 39] were the anti-virus chosen for testing[7]. First we examined if any of them detected/prevented the attack, later after the attack, we examined if any of them detected malicious software on the phone.

The results are that none of them prevented the attack at first. After the attack and after a scan had been performed, **AVG** detected that Androrat was installed and informed the user that it could be malicious. However, upon trying to uninstall the threat it states that it cannot, nothing more has been detected by **AVG**.

CM security after a scan had been performed, did not detect any vulnerabilities with the attack.

The other two virus scanner applications did not find anything wrong with the device.

None of them detected alterations to the `init.rc` file, or the `su` binary that was added, comparing to a previous state of the smartphone. This is a result of the lack of power that usual anti-virus apps have. Since most attackers exploit and pursue an attack vector that ends with having root access, which gives the attacker the upper hand against preventive applications such as these, with lower level access.

Because of this lack of power, such as these ones, even requesting all permissions available, have little room to detect/prevent attacks as this one. They have limited ability to scan certain areas of storage of the smartphone, like the root system folder, where the ramdisk is loaded at boot and the `init.rc` file is. Also they have very limited capability in removing the applications, like Androrat, because it is a system application.

4.5 USB Secure

Based upon the knowledge gathered with the research of vulnerabilities and of attacks over USB connection, we developed an Android application, which can safely let Android users charge their smartphones on public kiosks, without any fear of attacks. Unfortunately, the application needs root permission to activate the security mechanism

[7] All anti-virus were tested in 9th of June of 2014. AVG version was 4.1.1, Avast version was 3.0.7 ,CM security 1.6.1 and Virus Scanner 1.4.3

necessary, so the owner of the device needs a previously rooted the device.

The application is listening if any new USB connection is done and when it is, it prompts a warning activity to the foreground of the device, asking if the user trusts or not that connection.

If the user trusts the connection, the application does nothing and simply quits. In case the user does not trust, a series of security mechanisms will be triggered. There are two types of security mechanisms. Security mechanisms are platform independent, such as ADB enabled and mechanisms regarding the vulnerabilities found for specific products, which are platform dependent. Examples of these are vulnerabilities such as the ability to send AT commands over the USB connection.

4.5.1 Architecture

The application requires that the Android phone has at least version 2.2–2.2.3 Froyo (API level 8). And targets Android 4.3 Jelly Bean (API level 18).

To detect the USB connection, it was necessary to have a `BroadcastReceiver`. A `BroadcastReceiver` is basically a listener of intents that are broadcasted on the Android system. Intents are little messages that constitute the Android IPC, in the application layer. They could represent intention of an action, like launching an activity, or just general information about the phone state, like receiving a phone call, or the device as USB cable plugged. In order that a `BroadCastReceiver` receive the necessary intents, it has to be declared first in the `AndroidManifest.xml`, in the intent filter of that component. There it is registered the specific types of intents that are necessary that our `BroadcastReceiver` to listen to, in this case it is declared that intents are of `ACTION_POWER_DISCONNECTED` and `ACTION_POWER_CONNECTED`. Both represent when the external power has been connected or not.

When an external power has been connected, it will trigger the activity warning the user, asking if he trusts the USB connection. When the external power has been disconnected, it will reverse the mechanisms triggered.

Since the application is listening to the intents, `ACTION_POWER_DISCONNECTED` and `ACTION_POWER_CONNECTED`, it will prompt in an USB data connection or a simple

USB power charge.

4.5.1.1 Warning Activity

The warning activity is the main and only activity on the application. It issues a prompt when the user connects the USB cable, this way reminding the user that vulnerability may arise from a USB connection. It also provides availability so that the user does not have to search for the application at that moment.

The activity basically warns the user of the danger attached to that USB connection and asks to trust or not the connection. In the trusting case, nothing is done, but if the user does not trust the USB connection, a series of security mechanisms will be deployed. By default, the application triggers the security mechanism if a new USB connection plugs, only in the case the user does not trust the USB connection the security mechanisms will cease.

Figure 4.7: Interface of the warning activity

4.5.2 Security Mechanisms

There are two kind of security mechanisms, one that tackles general vulnerabilities. The other kind is security mechanisms that tackle vulnerabilities found on specific devices. The other kind is security mechanisms that tackle vulnerabilities found on specific devices.

To cope with future scalability that could arise from vulnerabilities attached with specific devices, we implement design patterns [40]. Therefore, when future updates are needed, an update could easily be developed.

The design pattern is a simple one, we use the factory method .The principal role of this pattern is that we have a “factory” that returns the correct object with the concrete implementations, according to the device model. The factory detects the device model and returns an object that implements the security mechanism for that device. That objects extends an abstract object with the abstract function for the platform dependent mechanism and implements a platform independent mechanism.

So with this approach for any new platform dependent vulnerabilities discovered, a new class implementing the mechanism for that platform can be created. And any new platform independent vulnerabilities discovered, only an abstract class is created. This way updating every object that extends that class.

So far, the following mechanisms are triggered:

- **Platform independent:** root privileges are used, to disable ADB connection.
- **Platform dependent:** Differs from platform to platform, so far only the device GT-s5839i is covered. For the GT-s5839i the main process that deals with the AT commands communication from USB to the modem is killed, which is the `dun_mgr`. After terminating this process, the `atx` process is terminated as well.

Since without root access further security prevention is impossible, for non-rooted phones we added a warning message, which shows the vulnerabilities that the phone may be in, as shown in 4.9.

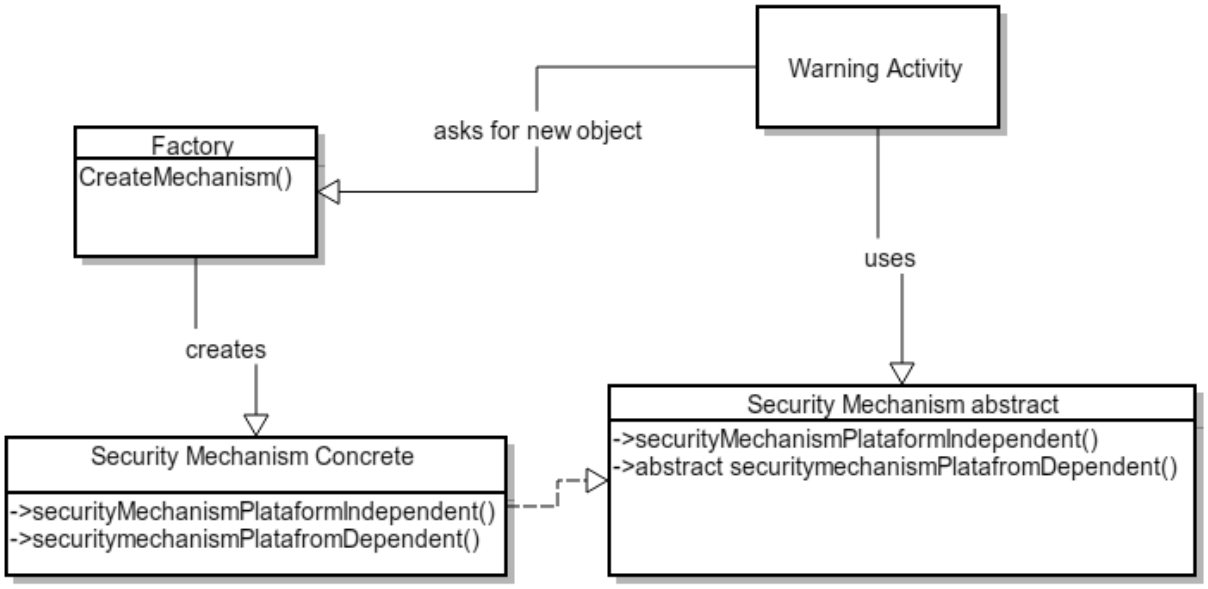


Figure 4.8: UML chart of the factory method implemented

Figure 4.9: Warning activity notifying user that it can not trigger the security mechanisms

Chapter 5

Conclusion

In this dissertation, we detail several USB vulnerabilities in Android and in the vendor customization of Android. In the research of vulnerabilities, we successfully unveil a new kind of attack by sending AT commands over USB, which is possible in some vendors' customization, to flash the phone with a compromised boot partition, with that, we were able obtain root access, install a surveillance application and make ADB always enabled. After that, the device is compromised.

Regarding the ability to send the AT commands over USB to Samsung devices, we believe it is not a bug in the development of the system, as these functionalities are intended to be used by the computer application of the vendor to configure and manage the smartphone. In our view, implementation of such "features" should be at least disclosed to users, in order that they understand the risks of an exposed USB connection. A comprehensive list of vulnerable devices is given, with regards to the vulnerabilities of the Samsung customization of devices. A series of anti-virus application are tested, to observe if any prevented the attack, or if after the attack occurs, they detect that the device is compromised and none of them prevents or detects the attack.

We give a practical attack scenario where such attack could be carried out, where normal users might be more prone to be infected. With that scenario in mind, we develop a proof of concept of the attack with characteristics that fits the attack scenario, like being fully automated.

Since the vulnerabilities are device depend, the proof of concept has been built with a

hierarchy of attacks, in which the aim of that hierarchy is to produce the maximum dose of attack, according to the device and to the vulnerabilities found.

Also with the success in attacking the device with such vulnerabilities, we were able to develop an application for Android, specially designed to prevent said attack. Unfortunately in order to completely mitigate the problem, the application needs to request root permission, in case it does not have it, it will just inform the users that it might be vulnerable.

5.1 Future work

Concerning the list of vulnerable devices, it is important to we continue to expand it by the experimenting with new devices, taking advantage of this, it would be important to notice as well, how many Samsung devices have an unlocked bootloader, since without it the attack could not happen.

We would like also to extend this attack to other manufacturers and find if any of them allow us to produce similar attacks. And if successful, we would like to add to our anti-virus application, security mechanisms to help mitigate the problem.

Dealing with the attack scenario, it would be an interesting to do a social experiment to find how many devices we could attack by hour, in a public charging kiosk. Experimenting in different places, different environments to determine which of those environments people are more prone to be infected.

Appendix A

Abbreviations

USB Universal Serial Bus

OS Operating System

ADB Android Debug Bridge

ADBD Android Debug Bridge Daemon

IPC Inter-Process communication

GPS Global Positioning System

SMS Short Message Service

AP Applications Processor

BP Baseband Processor

RIL Radio Interface Layer

NFC Near Field Communication

CVE Common Vulnerabilities and Exposures

API Application Programming Interface

MITM Man-in-the-Middle Attack

RFS Remote File System

RPC Remote Procedural Calls

VPN Virtual Private Network

CSC Country Sales Code

RAT Remote Access Tool

RILD Radio Interface Layer Daemon

References

[1] “Android filesystem config file,” accessed on 21/06/2014. [Online]. Available: https://android.googlesource.com/platform/system/core/+/master/include/private/android_filesystem_config.h

[2] T. Vidas, D. Votipka, and N. Christin, “All your droid are belong to us: A survey of current android attacks.” in WOOT, 2011, pp. 81–90.

[3] “CVE security vulnerability database. Security vulnerabilities, exploits, references and more,” accessed on 23/06/2014. [Online]. Available: http://www.cvedetails.com/

[4] C. Mulliner and C. Miller, “Injecting SMS messages into smart phones for security analysis,” in USENIX Workshop on Offensive Technologies (WOOT), 2009.

[5] H. Z. Zihang Xiao, Qing Dong and X. Jiang, “Oldboot: the first bootkit on Android,” 2014, accessed on 14/04/2014. [Online]. Available: http://blogs.360.cn/360mobile/2014/01/17/oldboot-the-first-bootkit-on-android/

[6] “Radio Layer Interface | Android Open Source,” accessed on 22/06/2014. [Online]. Available: http://www.kandroid.org/online-pdk/guide/telephony.html

[7] “Booting Android Bootloaders, fastboot and boot images,” accessed on 10/04/2014. [Online]. Available: http://2net.co.uk/slides/android-boot-slides-2.0.pdf

[8] Z. Wang and A. Stavrou, “Exploiting smart-phone USB connectivity for fun and profit,” Proceedings of the 26th Annual Computer Security Applications Conference on - ACSAC ’10, p. 357, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1920261.1920314

[9] C. Miller, “Exploring the nfc attack surface,” Proceedings of Blackhat, 2012.

[10] “Google Buys Android for Its Mobile Arsenal - Businessweek,” accessed on 10/04/2014. [Online]. Available: http://www.businessweek.com/stories/2005-08-16/google-buys-android-for-its-mobile-arsenal

[11] “Android Pushes Past 80% Market Share While Windows Phone Shipments Leap 156.0% Year Over Year in the Third Quarter, According to IDC - prUS24442013,” accessed on 10/04/2014. [Online]. Available: http://www.idc.com/getdoc.jsp?containerId=prUS24442013

[12] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, “The impact of vendor customizations on android security,” Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS ’13, pp. 623–634, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2508859.2516728

[13] R. Langner, “Stuxnet: Dissecting a cyberwarfare weapon,” Security & Privacy, IEEE, vol. 9, no. 3, pp. 49–51, 2011.

[14] “SanDisk Survey Shows Organizations at Risk from Unsecured Usb Flash Drives;Usage is More than Double Corporate IT Expectations,” accessed on 22/06/2014. [Online]. Available: http://www.sandisk.com/about-sandisk/press-room/press-releases/2008/2008-04-09-sandisk-survey-shows-organizations-at-risk-from-unsecured-usb-flash-drivesusage-is-more-than-double-corporate-it-expectations

[15] K. Andersson and P. Szewczyk, “Insecurity by obscurity continues: are adsl router manuals putting end-users at risk,” 2011.

[16] A. Pereira, M. E. Correia, and P. Brandão, “USB connection vulnerabilities on android smartphones: default and vendors’ customizations,” 5th Conference in the "Communications and Multimedia Security", 2014.

[17] A. Misra and A. Dubey, Android Security: Attacks and Defenses, 2013. [Online]. Available: http://books.google.pt/books/about/Android_Security.html?id=PNnRTu8tERIC&pgis=1

[18] J. J. Drake, Z. Lanier, C. Mulliner, P. O. Fora, S. A. Ridley, and G. Wicherski, Android Hacker’s Handbook. John Wiley & Sons, 2014. [Online]. Available:

http://books.google.com/books?id=xpc6AwAAQBAJ&pgis=1

[19] "System Permissions | Android Developers," accessed on 22/06/2014. [Online]. Available: http://developer.android.com/guide/topics/security/permissions.html

[20] "Security Enhancements in Android 4.3 | Android Developers," accessed on 12/04/2014. [Online]. Available: http://source.android.com/devices/tech/security/enhancements43.html

[21] "Signing Your Applications | Android Developers," accessed on 22/06/2014. [Online]. Available: http://developer.android.com/tools/publishing/app-signing.html

[22] "SamsungGalaxyBackdoor - Replicant Developers," accessed on 22/06/2014. [Online]. Available: http://redmine.replicant.us/projects/replicant/wiki/SamsungGalaxyBackdoor

[23] L. Jeter and S. Mishra, "Identifying and quantifying the android device users' security risk exposure," 2013 International Conference on Computing, Networking and Communications (ICNC), pp. 11–17, Jan. 2013. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6504045

[24] "Android Debug Bridge | Android Developers," accessed on 12/04/2014. [Online]. Available: http://developer.android.com/tools/help/adb.html

[25] "Cydia Impactor," accessed on 13/04/2014. [Online]. Available: http://www.cydiaimpactor.com/

[26] "SuperOneClick Root v2.3.3," accessed on 20/06/2014. [Online]. Available: http://www.superoneclickdownload.com/

[27] T. Specification and G. Terminals, "AT command set for 3GPP User Equipment (UE) (3G TS 27.007 version 2.0.0)," vol. 4, no. June, pp. 17–18, 1999.

[28] H. Welte, "Anatomy of contemporary GSM cellphone hardware," Unpublished paper, 2010.

[29] A. J. Singh and A. Bhardwaj, "Android Internals and Telephony," International Journal of Emerging Technology and Advanced Engineering, vol. 4, no. 1, pp. 51–59, 2014.

[30] J. F. Bluebox, "Android : One Root to Own Them All Please Complete Speaker

Feedback Survey Or else ," 2013.

[31] A. Hoog, Android forensics: investigation, analysis and mobile security for Google Android. Elsevier, 2011.

[32] "Android Init Language Google Git," accessed on 22/04/2014. [Online]. Available: https://android.googlesource.com/platform/system/core/+/master/init/readme.txt

[33] "Superuser rooting method," accessed on 12/04/2014. [Online]. Available: http://androidsu.com/superuser/

[34] "VRT: Androrat - Android Remote Access Tool," accessed on 12/04/2014. [Online]. Available: http://vrt-blog.snort.org/2013/07/androrat-android-remote-access-tool.html

[35] "Pywinauto - Windows GUI automation using Python." [Online]. Available: https://code.google.com/p/pywinauto

[36] "AVG AntiVirus FREE for Android," accessed on 9/06/2014. [Online]. Available: https://play.google.com/store/apps/details?id=com.antivirus

[37] "Avast Mobile Security Antivirus," accessed on 9/06/2014. [Online]. Available: https://play.google.com/store/apps/details?id=com.avast.android.mobilesecurity

[38] "CM Security," accessed on 9/04/2014. [Online]. Available: https://play.google.com/store/apps/details?id=com.cleanmaster.security

[39] "Virus Scan (Antivirus)," accessed on 9/06/2014. [Online]. Available: https://play.google.com/store/apps/details?id=com.pablosoftware.virusscan

[40] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design patterns: elements of reusable object-oriented software. Pearson Education, 1994.